C# / .NET

support de cours

ISTIA
2012-2013

Bertrand Cottenceau - bureau 311 ISTIA
bertrand.cottenceau@univ-angers.fr

Références bibliographiques :

Introduction au langage C# de Serge Tahé
disponible sur http://tahe.developpez.com/dotnet/csharp/

Transparents du cours de H.Mössenböck, University of Linz
http://www.ssw.uni-linz.ac.at/Teaching/Lectures/CSharp/Tutorial/

C# 4.0 in a Nutshell de Joseph et Ben Albahari (O'Reilly)

C# 3.0 Design Patterns de Judith Bishop (O'Reilly)

1. Introduction

Le langage C# (C Sharp) est un langage objet créé spécialement pour le framework Microsoft .NET. L'équipe qui a créé ce langage a été dirigée par *Anders Hejlsberg*, un informaticien danois qui avait également été à l'origine de la conception du langage *Delphi* pour la société Borland (évolution objet du langage Pascal).

Le **Framework .NET** est un environnement d'exécution (CLR Common Language Runtime) ainsi qu'une bibliothèque de classes (plus de 2000 classes). L'environnement d'exécution (CLR) de .NET est une machine virtuelle comparable à celle de Java.

Le runtime fournit des services aux programmes qui s'exécutent sous son contrôle : chargement/exécution, isolation des programmes, vérification des types, conversion code intermédiaire (IL) vers code natif, accès aux métadonnées (informations sur le code contenu dans les assemblages .NET), vérification des accès mémoire (évite les accès en dehors de la zone allouée au programme), gestion de la mémoire (Garbage Collector), gestion des exceptions, adaptation aux caractéristiques nationales (langue, représentation des nombres), compatibilité avec les DLL et modules COM qui sont en code natif (code non managé).

Les classes .NET peuvent être utilisées par tous les langages prenant en charge l'architecture .NET. Les langages .NET doivent satisfaire certaines spécifications : utiliser les mêmes types CTS (Common Type System), les compilateurs doivent générer un même code intermédiaire appelé MSIL (Microsoft Intermediate Language).

Le MSIL (contenu dans un fichier .exe) est pris en charge par le runtime .NET qui le fait tout d'abord compiler par le JIT compiler (Just In Time Compiler). La compilation en code natif a lieu seulement au moment de l'utilisation du programme .NET.

Définir un langage .NET revient a fournir un compilateur qui peut générer du langage MSIL. Les spécifications .NET sont publiques (Common Language Specifications) et n'importe quel éditeur de logiciel peut donc concevoir un langage/un compilateur .NET. Plusieurs compilateurs sont actuellement disponibles : C++.NET (version Managée de C++), VB.NET, C#, Delphi, J#

Lors du développement d'applications .NET, la compilation du code source produit du langage MSIL contenu dans un fichier .exe. Lors de la demande d'exécution du fichier .exe, le système d'exploitation reconnaît que l'application n'est pas en code natif. Le langage IL est alors pris en charge par le moteur d'exécution du framework .NET qui en assure la compilation et le contrôle de l'exécution. Un des points importants étant que le runtime .NET gère la récupération de mémoire allouée dans le tas (garbage collector), à l'instar de la machine virtuelle Java.

"En .NET, les langages ne sont guères plus que des interfaces syntaxiques vers les bibliothèques de classes."
Formation à C#, Tom Archer, Microsoft Press.

On ne peut rien faire sans utiliser des types/classes fournis par le framework .NET puisque le code MSIL s'appuie également sur les types CTS. L'avantage de cette architecture est que le framework .NET facilite *l'interopérabilité* (projets constitués de sources en différents langages). Avant celà, faire cohabiter différents langages pour un même exécutable imposait des contraintes de choix de types "communs" aux langages ainsi que de respecter des conventions d'appel de fonctions pour chacun des langages utilisés.

Remarque (particularité de C++.NET) : la version .NET de C++ a la particularité de permettre à l'utilisateur de générer soit du code natif soit du code managé. Avec Visual C++ Express, le langage C++ a été complété pour permettre la gestion managée de la mémoire allouée dans le tas.

2. Les héritages du langage C

Le C# reprend beaucoup d'éléments de syntaxe du C. Partant du langage C, beaucoup d'éléments de syntaxe sont familiers :

- structure des instructions similaires (terminées par ;) : déclaration de variables, affectation, appels de fonctions, passage des paramètres, opérations arithmétiques
- blocs délimités par {}
- commentaires // ou /* */
- structures de contrôle identiques : if/else, while, do/while, for(; ;)
- portée des variables : limitée au bloc de la déclaration

En C#, on peut déclarer une variable n'importe où avant sont utilisation. Ce n'est pas nécessairement au début d'une fonction.

```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 4; i++)
            {
                Console.WriteLine("i={0}", i);
            }
        }
    }
}
```

2.1. Premier exemple console

Tout ne peut pas être expliqué dans un premier temps. **System.Console** est la classe de gestion des I/O en mode console.

Point d'entrée et entrées/sorties en mode console

La fonction principale (point d 'entrée) est la fonction statique **`void Main(...)`**. En mode console, les entrées sorties (saisies clavier + affichages) se font grâce à la classe **`Console`**.

```
class Program
{
        public static void Main()
        {
            int var=2;
            string str="toto";
            Console.WriteLine("var= {0} str = {1}", var, str);
            str = Console.ReadLine();
            Console.WriteLine("Ligne lue = {0}", str);
        }
}
```

Un second exemple où l'on voit les similitudes avec la syntaxe du C.

```
using System;
namespace ConsoleApplication
{
    class Program
    {
```

```
        static void Main(string[] args) // Point d'entrée
        {
            int var = 2;
            for (int i = 0; i < 5; i++)
            {
                var += i;
                Console.WriteLine("i={0} var={1}", i, var);
            }

            float f=2F;
            while (f < 1000F)
            {
                f = f * f;
                Console.WriteLine("f={0}", f);
            }
        }
    }
}
```

```
i=0 var=2
i=1 var=3
i=2 var=5
i=3 var=8
i=4 var=12
f=4
f=16
f=256
f=65536
Appuyez sur une
touche pour
continuer...
```

Les arguments de la ligne de commande

Lorsque l'on exécute un programme console depuis l'invite de commandes, on peut passer des paramètres qui sont reçus en arguments de la fonction statique Main().

```
E:\Enseignement\C#>Prog Toto 23 Texte
Programme appelé avec les paramètres suivants:
argument n° 0 [Toto]
argument n° 1 [23]
argument n° 2 [Texte]
```

```
static void Main(string[] args)
{
          Console.WriteLine("Programme appelé avec les paramètres suivants:");
          for (int i = 0; i < args.Length; i++)
          {
                    Console.WriteLine("argument n° {1} [{0}]", args[i],i);
          }
}
```

2.2. Types primitifs du C# (alias de types .NET)

Le framework .NET fournit les types utilisables, et ce quel que soit le langage utilisé (VB.NET, C#, C++.NET). Dans C#, les types élémentaires sont nommés comme suit

`char` : caractère Unicode 16bits
`sbyte`/**`byte`** : entier 8 bits signé/non signé
`short`/**`ushort`** : entier 16 bits signé/non signé
`int`/**`uint`** : entier 32 bits signé/non signé,
`long`/**`ulong`** : entier 64 bits signé/non signé
`float` : flottant 32 bits,
`double` : flottant 64 bits ,
`decimal` (128 bits) : entier multiplié par une puissance de 10
`string` : chaîne de caractères. Type référence qui s'utilise comme un type valeur.

2.3. Littérals (constantes) / Format des constantes

En C#, les constantes numériques peuvent presque être considérées comme typées. Selon la notation utilisée

pour décrire une constante, le codage n'utilise pas le même nombre de bits.

Constantes entières

```
int x=10;            // x ppv 10 (base 10)
int x = 0x2F;        // x ppv (2F)h
long l= 10000L;      // le L signifie que la constante est sur 64 bits
```

Constantes réelles

Par défaut, les constantes réelles sont codées sur 64 bits. Pour que les constantes réelles soient codées sur 32 bits, il faut ajouter un suffixe F. Pour les constantes décimales, il faut ajouter un suffixe m.

```
double     d1=10.2;     //OK
float      f1=10.2;     //Error
float      f2=10.2F;    // OK
double     d2=1.2E-2;   // d2 = 0,012
float      f3 = 3.1E2F; // f3=310
decimal    d4=2.5m;
```

Constantes chaînes de caractères

```
string     s1="abc";
```

Chaîne verbatim **:** lorsque la chaîne est préfixée par @, aucun des caractères entre les guillemets n'est interprété. Pour représenter les " dans une telle chaîne, il faut les doubler

```
string     s2=@"a\nebc";
string     s3=@"a\n""ebc";      // s ppv a\n"ebc
```

Caractères de contrôle

`\n (newline) \t (horizontal tab) \v (vertical tab) \b (backspace) \r (carriage return) \f (form feed) \\ (backslash) \' (single quote) \" (double quote) \0 (caractere de valeur 0).`

2.4. Dépassements

Par défaut, les dépassements numériques sont silencieux. On n'est pas informé si celà se produit.

On peut néanmoins tester si les opérations arithmétiques ne conduisent pas à des dépassements, ou si les casts sont dégradants (est-ce que la conversion long → int est sans perte de données). Les opérations réalisées dans le bloc **`checked`** conduisant à un dépassement déclenchent une exception (cf. Exceptions)

```
try
  {
      checked
      {
          long l = 1000;
          l = l * l;
          l = l * l;
          l = l * l;
          Console.WriteLine(l);
      }
  }
  catch (Exception e)
  {
      Console.WriteLine(e);
  }
```

2.5. Les fonctions et les passages de paramètres (ref, out)

En C#, les fonctions sont nécessairement définies au sein de types structurés (struct ou class). Il n'y a pas réellement de fonction "globale" au sens du C. Néanmoins, les fonctions membres suivent la même logique de définition et d'appel que les fonctions du C. La principale différence est que le compilateur C# n'a pas besoin de fichier de prototypes de fonction (les fichiers .h header du C). Il retrouve seul la signature de la fonction à l'intérieur de données de description de type. La définition des fonctions suffit. Il n'y a donc pas de fichiers d'entête (.h) ce qui élimine beaucoup de problèmes de compilation.

En C#, il n'y a pas de prototype pour les fonctions (différent du C)
Le passage des paramètres se fait par valeur (comme en C)

```
using System;

namespace ConsoleApplication
{
    class Program
    {
        static int f1(int param)
        {
            return 2*param;
        }

        static void Main(string[] args)
        {
            Console.WriteLine("f1(3)={0}", f1(3));
        }
    }
}
```

Les paramètres modifiables : mots clé ref et out

Par défaut, les paramètres sont passés par valeur, comme en C. Celà signifie que la fonction reçoit une copie du paramètre passé à l'appel. Les éventuelles modifications du paramètre dans la fonction n'ont aucune incidence sur la valeur du paramètre fourni lors de l'appel de la fonction.

Passage par valeur

```
class Program
    {
        static int f1(int param)  // param est initialisé à partir de x
        {
            param++;              // param est incrémenté, mais sans effet sur x
        }

        static void Main(string[] args)
        {
            int x=2;
           f1(x);
           Console.WriteLine("x={0}", x);   // x vaut toujours 2
        }
    }
```

Le C# introduit des mots clés pour que des paramètres d'appel puissent être modifiables dans une fonction (ca reprend l'idée du Pascal avec le mot clé var ou des références en C++). Les mots clés utilisés en C# sont

`ref` : indique que le paramètre formel est un alias du paramètre d'appel. C'est à dire c'est la même variable mais avec un autre nom localement.

`out` : idem. Ce paramètre n'est pas utilisé en lecture, seulement en écriture.

```
using System;

namespace ConsoleApplication
{
    class Program
    {
        static void Inc(ref int EntreeSortie) // le paramètre est une entrée sortie
        {                                     // paramètre en lecture écriture
            EntreeSortie++;
        }

        static void Reset(out int Sortie)   // le paramètre est en écriture seulement
        {
            Sortie=0;
        }

        static void Main(string[] args)
        {
            int x = 2;
            Inc(ref x);         // Il faut remettre le mot clé ref pour l'appel !
            Console.WriteLine(x);
            Reset(out x);       // Il faut remettre le mot clé out pour l'appel !
            Console.WriteLine(x);
        }
    }
}
```

2.6. Définition d'un type structuré (similaire aux structures du C)

Le mot clé **`struct`** permet la définition d'un type. Le type créé représente des variables structurées avec un ou plusieurs champs de données.

Ci dessous, une variable de type **`PointS`** est une variable contenant deux champs entiers. Ces champs s'appellent **`X`** et **`Y`** et peuvent être désignés à l'aide de la notation **`p.X`** et **`p.Y`** où **`p`** est une variable de type **`PointS`**. C'est très proche des types structurés du langage C.

La nouveauté est que l'on doit préciser un niveau **d'accessibilité** (ici les champs sont publics).

```
using System;

namespace Exemple
{
    struct PointS   // PointS est un type
    {
        public int X;
        public int Y;
    } // fin PointS

    class Program
    {
        static void Main(string[] args)
        {
            PointS p1,p2;  // p1 et p2 sont des variables de type PointS
            p1.X = 3;
            p1.Y = 4;

            p2.X = 6;
            p2.Y = 1;
        }
    }
}
```

2.7. Type structuré (struct) avec des fonctions membres

Du fait de l'orientation objet du langage C#, on peut avoir des fonctions membres dans les types structurés, et pas uniquement des données. Il suffit de définir des fonctions au sein des blocs décrivant le type structuré. Dès lors, sur une variable, on peut appeler des fonctions rattachées à cette variable. La notation est homogène avec l'accès aux champs de la structure. On remarquera d'ailleurs que les fonctions utilisées sont nécessairement membres d'un type : soit struct soit class. De plus, on peut avoir plusieurs fonctions membres ayant le même nom, mais des signatures différentes. C'est ce qu'on appelle la surcharge.

```
using System;
namespace Exemple
{
    struct PointS
    {
        public int X;
        public int Y;

        public void SetXY(int pX, int pY)
        {
            X = pX;
            Y = pY;
        }

        public int GetX()       // les deux fonctions membres GetX ont des signatures
        {                       // différentes
            return X;
        }
        public void GetX(ref int pX)
        {
            pX = X;
        }

        public string ToString()
        {
            return "(" + X.ToString() + "," + Y.ToString() + ")";
        }
    } // fin PointS

    class Program
    {
        static void Main(string[] args)
        {
            PointS p1 = new PointS();   // p1 est une variable de type PointS
            PointS p2 = new PointS();   // p2 est une variable de type PointS

            p1.SetXY(2, 3);     // appel de la fonction membre SetXY sur la variable p1
            p2.SetXY(6, 1);

            Console.WriteLine("p2.X={0}", p2.GetX());
            Console.WriteLine("p1" + p1.ToString());
            Console.WriteLine("p2" + p2.ToString());
        }
    }
}
```

```
Sortie console
p2.X=6
p1(2,3)
p2(6,1)
```

3. La programmation orientée objet en C#

3.1. Le paradigme objet en bref (vocabulaire)

La Programmation Orientée Objet (POO) est une façon de programmer et de décomposer des applications en informatique. Un ***objet*** est une ***variable structurée*** contenant des champs mais aussi des fonctions membres. Ceci est possible en C# mais ne l'est pas en langage C.

Qu'est-ce qu'un objet ? C'est une variable structurée avec des fonctions membres. Les variables de la structure **`PointS`** (vue précédemment) sont en quelque sorte déja des objets.

Les attributs : ce sont les champs de données d'un objet. C'est une des caractéristiques d'un objet. Ils décrivent les données nécessaires pour décrire un objet. Par exemple, **`X`** et **`Y`** sont les attributs d'un objet de type **`PointS`**

Les méthodes : autre nom (utilisé en POO) pour les fonctions membres. Ces fonctions membres représentent les mécanismes/ les fonctionnalités dont dispose un objet. C'est ce qu'il sait faire, ça décrit son *comportement*. L'ensemble des méthodes d'un objet est aussi appelé *l'interface d'un objet.*

L'encapsulation : en POO, les attributs ne sont généralement pas accessibles directement (contrairement à X et Y dans le type PointS). On va pouvoir modifier le niveau d'accès des membres (mots clés d'accessibilité) de façon à ce que les données d'un objet (valeurs de ses attributs) ne soient pas accessibles directement, mais uniquement grâce à ses méthodes. Le fait de cacher certains détails de fonctionnement d'un objet est appelé ***encapsulation***. Les mécanismes de manipulation des données d'un objet sont généralement cachés à l'utilisateur. Ceci de façon à ce que l'utilisateur ne se concentre que sur les méthodes de l'objet (ce qu'il sait faire). Le gros intérêt de l'encapsulation est que l'on peut très bien modifier le comportement interne d'un objet (par exemple certains algorithmes) sans changer son comportement global vu de l'extérieur : l'objet rend les mêmes services mais avec des algorithmes différents. On arrive ainsi à découpler certaines parties du logiciel.

Etat d'un objet : c'est la valeur de l'ensemble de ses attributs à un instant donné. Par exemple, l'état d'un objet de type **`PointS`** est simplement la valeur des coordonnées.

3.2. Le mot clé class (class vs struct)

Le C# (comme d'autres langages objet) utilise le mot clé **`class`** pour pouvoir définir d'autres types structurés avec des méthodes (**`struct`** ne convient pas à tous les objets). Les types **`class`** sont plus riches que les types **`struct`** (on peut dériver des classes) et conduisent également à un fonctionnement différent de la gestion de la mémoire. On peut donc aussi définir un type class pour représenter des points du plan.

```
using System;
namespace TypeClass
{
    class Program
    {
        class PointC
        {
           // donnés membres privées (non accessibles depuis l'extérieur de l'objet)

            private int X;  // les attributs X et Y sont encapsulés
            private int Y;

           // les méthodes sont publiques

            public void SetXY(int pX,int pY)
            {
                X=pX;
                Y = pY;
            }

            public void GetXY(out int pX, out int pY)
            {
                pX = X;
                pY = Y;
            }

            public string ToString()
            {
                return "(" + X.ToString() + "," + Y.ToString() + ")";
            }
        } // fin PointC
```

```
        static void Main(string[] args)
        {
            PointC p3 = new PointC();
            PointC p4 = new PointC();

            p3.SetXY(2, 3);
            p4.SetXY(6, 1);
            Console.WriteLine(p3.ToString());
            Console.WriteLine(p4.ToString());

            int x, y;
            p3.GetXY(out x, out y);
            Console.WriteLine("x={0} y={1}",x,y);

           // p3.X = 4; // IMPOSSIBLE l'attribut X de l'objet p3 n'est pas accessible
        }
    }
}
```

```
(2,3)
(6,1)
x=2 y=3
Appuyez sur une touche pour continuer...
```

Qu'est-ce que l'instanciation ? C'est le fait de créer une instance (ç-à-d un objet) d'un type class ou struct. C'est donc le fait de créer une nouvelle variable objet. Un type classe décrit ce qu'ont en commun tous les objets qui seront générés, ç-à-d leurs caractéristiques communes. Un objet est donc une instance particulière d'un ensemble potentiellement infini d'objets ayant des traits communs : mêmes méthodes et mêmes attributs. En revanche les objets n'ont pas tous le même état (les attributs n'ont pas tous les mêmes valeurs !)

Dans l'exemple précédent, p3 et p4 désignent deux instances (deux objets) de la classe **`PointC`**. Les deux objets ont les mêmes attributs et les mêmes méthodes (les mêmes caractéristiques). Mais ils n'ont pas le même état.

3.3. Types valeurs / Types référence

En C#, selon le type d'une variable / d'un objet (rappelons qu'un objet n'est rien d'autre qu'une variable structurée), l'allocation de mémoire est différent. L'allocation d'une variable n'est donc pas le simple fait du programmeur (comme en C où l'on choisit ce que l'on met dans la pile (par défaut) et ce que l'on met dans le tas (allocation dyamique par des fonctions spécifiques)).

En C#, selon le type d'une variable, celle-ci est réservée soit dans la pile soit dans le tas (mémoire allouée dynamiquement). On distingue deux catégories de types : les types "valeur" et les types "référence".

Les types "valeur" → alloués dans la pile

Les types "référence" → alloués dans le tas (allocation dynamique)

Quels sont les types "valeur" ? Tous les types numériques primitifs du C# (byte, char, long …), les énumérations et les variables générées à partir d'un type **`struct`**. Le type structuré **`PointS`** est un type valeur. Quand on crée une variable de type **`PointS`**, elle est réservée dans la pile. La mémoire des variables de type valeur est libérée automatiquement à la fin du bloc de leur déclaration (comme les variables locales en C)

Quels sont les types "référence" ? Les types créés avec le mot clé **`class`** (ainsi que le type string).

Dans ce cas, la mémoire est allouée *dynamiquement* dans le tas quand on crée un objet (avec le mot clé new). Quand on instancie une classe (ie qu'on crée un objet), on réserve de la mémoire dans le tas. On récupère une référence (on peut voir ca comme un pointeur) sur l'emplacement réservé. Pour les types références, ce sont des références qui permettent de désigner les objets.

Quand la mémoire est-elle désallouée ? : pour les variables de type valeur, à la fin du bloc de déclaration. Pour les objets de type référence, quand il n'y a plus aucune référence sur un objet donné (il n'y a alors plus aucun moyen d'y accéder). Dans ce cas, le Garbage Collector (un programme qui s'exécute périodiquement pour libérer de la mémoire) récupère la mémoire de l'objet. Ca veut dire que la zone mémoire peut de nouveau être allouée à un autre objet nécessaitant de la mémoire dans le tas. L'intérêt du C# par rapport au C/C++, est qu'on ne peut donc pas oublier de désallouer de la mémoire (fuites mémoires). La désallocation n'est pas de notre responsabilité.

3.4. Distinctions Types valeur/Types référence concernant l'affectation

Cette distinction entre types “valeur” et types “référence” est essentielle en .NET. C'est un as^pect du système qu'il faut vraiment bien comprendr. De plus, l'affectation n'a pas le même sens pour les variables de type valeur et celles de type référence.

Affectation pour les types “valeur” Une variable de type valeur contient des données.

```
          int i=18;
// i est une variable allouée dans la pile contenant l'entier signé 18 codé sur 32 bits
```

Remarque : même dans la situation ci-dessous, la variable est dans la pile.

```
          int i = new int();   //i est néanmoins alloué dans la pile
```

Pour les variables de type valeur, l'affectation réalise l'affectation des valeurs.

```
          int a=12,b=4;    // a contient 12 et b contient 4
          a=b;             // a contient désormais la valeur 4.
                          // la valeur de b a été affectée à la variable a
```

Même pour les variables de type struct, l'affectation affecte la valeur, champs par champs. Dans l'exemple ci-dessous où **`PointS`** est le type struct défini précédemment, l'affectation **`p2=p1`** réalise l'affectation attribut par attribut.

p2=p1 équivaut à p2.X=p1.X et p2.Y=p1.Y (même si les champs X et Y étaient privés!)

```
   static void Main(string[] args)
        {
            PointS p1 = new PointS();  // p1 et p2 dans la pile
            PointS p2 = new PointS();     // même si on écrit new PointS()

            p1.SetXY(2, 3);  // p1 a l'état (2,3)

            p2 = p1;         // l'état (2,3) est affecté à p2

            Console.WriteLine("p2 = " + p2.ToString());   // p2 = (2,3)
        }
```

Affectation pour les types référence

Une variable de type référence est une variable pour laquelle l'accès se fait via une référence. Tous les objets des classes sont manipulés ainsi. L'allocation de ces variables est faite dans le tas et leur désallocation est gérée par le garbage collector de .NET.

```
PointC p3;           // p3 est une variable qui peut faire référence à un objet de type
                     // PointC
p3 = new PointC();   // alloue un objet dans le tas et affecte la référence à p3
```

Dans le cas des variables de type référence, le mot clé new indique que l'on demande à créer une nouvelle instance. Celle-ci est alors allouée dans le tas (mémoire allouée dynamiquement).

Dans le cas des variables de type référence, on peut voir les références (les variables qui désignent les objets) comme des sortes de pointeurs sécurisés. Par exemple, une référence qui ne désigne aucun objet vaut la valeur **`null`**. C'est la valeur par défaut d'une référence non initialisée.

```
          PointC p3 = null;  // p3==null signifie "ne désigne aucun objet"
```

```
static void Main(string[] args)
{
  PointC p3;          // p3 vaut null
  PointC p4;          // p4 vaut null

   p3 = new PointC();  // p3 désigne une instance PointC
  p4 = new PointC();   // p4 désigne une autre instance

  p3.SetXY(2, 3);
  p4.SetXY(6, 1);
}
```

ATTENTION! Pour les variables de type référence, l'affectation réalise seulement une copie des références (comparable à une copie de pointeurs en C++). Dans l'exemple ci-dessous, quand on affecte p3 à p4, puisque **PointC** est un type référence, les variables p3 et p4 sont des références. Par conséquent, après l'affectation p4=p3, p4 fera désormais référence au même objet que p3. Il y aura alors deux références au même objet (celui d'état (2,3)). L'objet dont l'état est (6,1) n'aura donc plus de référence qui le désigne. Il n'est plus référencé et le Garbage Collector peut donc désallouer sa mémoire. Mais cette désallocation n'est pas instantanée.

```
static void Main(string[] args)
{
          PointC  p3 = new PointC();      // une instance
          PointC  p4 = new PointC();      // une seconde

          p3.SetXY(2, 3);       // l'instance désignée par p3 vaut (2,3)
          p4.SetXY(6, 1);       // celle désignée par p4 vaut (6,1)

          p4=p3;     // p4 désigne désormais le même objet que p3
                     // c'est à dire l'instance d'état (2,3)

          // l'instance dont l'état est (6,1) n'est plus accessible !
          // le Garbage Collector peut libérer sa mémoire
}
```

3.5. Distinction type valeur/type référence concernant l'égalité (==)

L'opérateur == pour les types valeur

Par défaut (puisque l'opérateur == peut être redéfini), le test d'égalité avec l'opérateur == sur des types valeur teste l'égalité de valeur des champs. Il y a comparaison de la valeur de tous les champs. L'opérateur renvoie la valeur true seulement si tous les champs ont les mêmes valeurs deux à deux

L'opérateur == pour les types référence

Par défaut, le test d'égalité avec l'opérateur == teste normalement si deux références désignent le même objet. Mais puisque l'opérateur == peut être reprogrammé, il peut très bien réalisé un traitement différent. Par exemple, pour les chaînes de type string, qui est pourtant un type référence, le test avec l'opérateur == indique l'égalité des chaînes (même longueur et mêmes caractères).

4. L'écriture de classes en C# (types référence)

4.1. Namespaces

Le mot clé **namespace** permet de définir un espace de nommage. Ceci permet de structurer le nommage des types (on peut rattacher un type à un espace de noms) et de lever d'éventuels conflits de noms. On peut avoir un même nom de type dans différents namespaces. Ci-dessous, on définit une classe **A** dans différents namespaces. Il s'agit donc de classes différentes.

Les namespaces peuvent être imbriqués. Le nom complet du type **A** du namespace **EspaceA** est **EspaceA.A**.

```
using System;

namespace EspaceA  // namespace avec deux types A et B
{
    class A { }
    class B { }
}

namespace EspaceB  // namespace avec un espace de nommage imbriqué
{
    namespace ClassesImportantes
    {
        class A { }
    }
}
namespace EspaceB.Divers  // equivaut à namespace EspaceB{ namespace Divers{... } }
{
    class A { }
}
namespace EspaceB.ClassesImportantes  // on complète le namespace
{
    class B { }
}
namespace ExempleNamespace  // namespace de la classe qui contient Main
{
    class Program
    {
        static void Main(string[] args)
        {
            EspaceA.A p1 = new EspaceA.A();
            EspaceB.ClassesImportantes.A p2 = new EspaceB.ClassesImportantes.A();
            EspaceB.Divers.A p3 = new EspaceB.Divers.A();
           EspaceB.ClassesImportantes.B p4;
        }
    }
}
```

Directive using Cette directive permet d'importer un namespace. Celà signifie que s'il n'y a pas d'ambiguité, les noms de types seront automatiquement préfixés par le namespace.

Par exemple, la classe **Console** de gestion des entrées sorties console est définie dans le namespace **System**. Le nom complet de cette classe est donc **System.Console**. L'utilisation devrait donc s'écrire

```
            System.Console.WriteLine("Hello!");
```

Ce qui signifie *"appel de la méthode `WriteLine` du type `Console` du namespace `System`"*. En important ce namespace (en début de fichier) avec le mot clé **using**, on peut simplifier l'écriture

```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           Console.WriteLine("Hello!");     // le type Console est recherché dans
                                            // le namespace System
        }
    }
}
```

4.2. Accessibilité

Les membres définis au sein des types peuvent avoir un niveau d'accessibilité parmi

`public` : membre (donnée ou fonction) accessible par tous

`private` : membre accessible seulement par une méthode d'un objet de la classe

`protected` : membre accessible par une méthode d'un objet de la classe ou par une méthode d'un objet d'une classe dérivée (voir plus tard la dérivation)

C'est grâce à ces modificateurs d'accès que l'on assure l'encapsulation des données. Tous les membres (attributs, méthodes ...) doivent être déclarés avec un niveau d'accessibilité. Par défaut, un membre est privé.

Il y a aussi des modificateurs d'accès spécifiques (non expliqué dans ce support)

`internal` : accès autorisé pour l'assemblage courant
`protected internal` : assemblage courant + sous classes

```
class MaClasse
{
     private int _valeur;
     public MaClasse(int v)
     {
        _valeur=v;
     }
     public void Affiche()
     {
        Console.WriteLine(_valeur);
     }
}
```

Ci-dessous un exemple complet avec la définition de la classe **`MaClasse`** et son utilisation dans le programme. Notons qu'il est possible aussi de donner une initialisation des attributs pour les types **`class`**. Ce n'est pas autorisé pour les types **`struct`**.

```
using System;
namespace SyntaxeCSharp
{
    class MaClasse
    {
          private int _valeur =2;  // initialiseur de champs
          public MaClasse(int v)
          {
                    _valeur = v;
          }


          public void Affiche()
          {
            Console.WriteLine("Objet MaClasse = {0}",
                                    _valeur);
          }
    }
    class Program
    {
        static void Main(string[] args)
        {   MaClasse m1 = new MaClasse(13);
            m1.Affiche();
            MaClasse m2 = m1;   // 2ième référence à l'objet
            if (m2.Equals(m1)) Console.WriteLine("Mêmes Objets");
        }
    }
}
```

```
Sortie Console
Objet MaClasse = 13
Mêmes Objets
Appuyez sur une touche pour
continuer...
```

4.3. Surcharge des méthodes

Il est possible de *surcharger les méthodes*, c'est-à-dire définir plusieurs méthodes ayant le même nom mais des signatures différentes, c'est-à-dire

- dont le nombre de paramètres est diffèrent

- le type des paramètres est différent
- les paramètres peuvent aussi avoir même type même nombre mais différer du mode de passage de paramètre (par valeur ou par référence)

En revanche, le type de retour d'une fonction ne fait pas partie de la signature. On ne peut pas déclarer deux fonctions qui ne diffèrent que par le type de retour.

```
 class A
 {
    private int _val;
    public int Val(){ return _val; }

    public void Val(int v){_val = v;  }

    public void Val(out int v){ v = _val;}
 }
class Programme
{
    static void Main()
    {
        A refA = new A();

        refA.Val(2);
        int v;
        refA.Val(out v);
        Console.WriteLine(refA.Val());
    }
}
```

4.4. Champs constants / en lecture seule

```
        class MaClasse
        {
                public const int _valC=12;      //par défaut statique
        }
```

Un littéral (constante) doit être utilisé pour initialiser un champ constant.

```
        class MaClasse
        {
                public readonly int _valRO=12;
        }
```

La valeur d'un attribut **`readonly`** peut être fixée par le constructeur. C'est donc une constante dont l'initialisation de la valeur peut être retardée jusqu'au moment de l'exécution (et non à la compilation).

```
        class MaClasse
        {
                public readonly int _valRO;
                public MaClasse(int val)
                {
                        _valRO=val;
                }
        }
```

4.5. Objet courant (référence this)

Dans une méthode, on appelle "Objet Courant", l'objet sur lequel a été appelée la méthode.

Dans l'exemple ci-dessous, quand on invoque **`obj1.Compare(obj2)`**, dans la méthode `Compare`, l'objet courant est **`obj1`**. Dans une méthode, on peut désigner l'objet courant par la référence **`this`**.

Notons que par défaut dans une méthode, quand on désigne un attribut (sans préfixer par une référence d'objet), il s'agit d'un attribut de l'objet courant. Sauf si une variable locale porte le même nom qu'un attribut, ce qui est déconseillé dans la mesure où c'est source d'erreur.

```
using System;
namespace ObjCourant
{
    class MaClasse
    {
        private int _val;
        public MaClasse(int val)
        {
            _val = val;
        }

        public bool Compare(MaClasse obj)
        {
           // on distingue ici l'attribut de l'objet courant (_val) de
           // l'attribut obj._val.
            return _val == obj._val;  // équivaut à this._val==_obj.val
        }
    }
    class Program
    {
        static void Main()
        {
            MaClasse obj1 = new MaClasse(12);
            MaClasse obj2 = new MaClasse(14);

            Console.WriteLine("obj1 et obj2 ont le même état = {0}", obj1.Compare(obj2));

        }
    }
}
```

4.6. Constructeurs (initialisation des objets)

Un *constructeur* est une méthode qui initialise l'état d'un objet (ne retourne rien) juste après son instanciation. Il a pour rôle d'éviter qu'un objet ait un état initial indéfini. Comme pour toutes les méthodes, il peut y avoir surcharge des constructeurs (c'est-à-dire plusieurs constructeurs avec des signatures différentes). De même qu'en C++, un constructeur a comme nom le nom de la classe.

Un constructeur peut recevoir un ou plusieurs arguments pour initialiser l'objet. Ces arguments sont fournis au moment de l'instanciation avec le mot clé new.

Remarque : si l'on ne met aucun constructeur dans une classe, il y en a alors un sans paramètre fourni par le compilateur (ce mécanisme est mis en place par défaut). Ce constructeur fourni par défaut ne fait aucun traitement mais permet d'instancier la classe. Dès lors qu'au moins un constructeur est spécifié, le constructeur sans paramètre par défaut n'est plus mis en place.

```
using System;
namespace SyntaxeCSharp
{
    class MaClasse
    {
         private string _nom;
         private int _val;
         public MaClasse()  // constructeur sans argument
         {
                _nom = "Nestor";
                _val = 12;
         }
```

```
        public MaClasse(int val) // constructeur avec un argument de type int
        {
               _val = val;
               _nom = "Nestor";
        }
        public MaClasse(int val, string nom){
               _val = val;
               _nom = nom;
        }
    } // fin MaClasse

    class Program
    {
        static void Main(string[] args)
        {
            MaClasse m1 = new MaClasse();           //m1 ->  ("Nestor",12)
            MaClasse m2 = new MaClasse(23);         //m2 ->  ("Nestor",23)
            MaClasse m3 = new MaClasse(17, "Paul"); //m3 ->  ("Paul",17)
        }
    }
}
```

4.6.1. Un constructeur peut appeler un autre constructeur (mot clé this)

On peut utiliser un autre constructeur dans un constructeur. Celà permet de bénéficier d'un traitement d'initialisation déja fourni par un autre constructeur. La notation utilise également le mot clé this :

```
        public MaClasse(int val):this()
        {

        }
```

Un exemple ci-dessous.

```
using System;
namespace SyntaxeCSharp
{
        class MaClasse
        {
              private  string _nom;
              private  int _val;
              public MaClasse()
              {
                    _nom = "Nestor";
                    _val=12;
              }
              public MaClasse(int val):this() // appel du constructeur sans paramètre
              {                               // avant ce bloc
                    _val=val;
              }

              /* ce constructeur utilise celui avec 1 paramètre, qui lui même
               invoque celui sans paramètre*/
              public MaClasse(int val,string nom):this(val)
              {
                    _nom=nom;
              }
        }
    class Program
    {
        static void Main(string[] args)
        {
           MaClasse obj=new MaClasse(20);           //obj._nom="Nestor" obj._val=20
        }
    }
}
```

4.7. Membres statiques (données ou méthodes)

Une *donnée membre statique* (appelée aussi *variable de classe*) n'existe qu'en un seul exemplaire quel que soit le nombre d'objets créés (même si aucun objet n'existe).

Une *fonction membre statique* (appelée aussi *méthode de classe*) n'accède pas aux attributs des objets. En revanche, une fonction membre statique peut accéder aux données membres statiques. Une telle fonction peut donc être invoquée même si aucune instance n'existe. Ce mécanisme remplace les fonctions hors classes du C++. Par exemple, les fonctions mathématiques sont des fonctions statiques de la classe **`System.Math`**.

```
  Console.WriteLine(System.Math.Sin(3.14/2));
```

```
using System;
namespace SyntaxeCSharp
{
   class MaClasse
    {
        static private int _nbObjetsCrees = 0;
        static public int GetNbObjets()
        {
            return _nbObjetsCrees;
        }
        public MaClasse()
        {
            _nbObjetsCrees++;
        }
     } // MaClasse

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(MaClasse.GetNbObjets());
            MaClasse m = new MaClasse();
            Console.WriteLine(MaClasse.GetNbObjets());
        }
    }
}
```

4.7.1. Constructeur statique

Constructeur appelé une fois avant tout autre constructeur. Permet l'initialisation des variables de classe (champs statiques.) Il ne faut pas donner de qualificatif de visibilité (public/protected/private) pour ce constructeur.

```
using System;
namespace SyntaxeCSharp
{
   class MaClasse
       {
            private int _var=12;
            private string _str;
            static private int _stat;

            static MaClasse(){
                 _stat = 12;
            }
            public MaClasse(string s){
                 _str=s;
            }
            public void Affiche()
            {
                 Console.WriteLine("({0},{1}",_str,_var,_stat);
            }
       }
```

```
        class Program
        {
         static void Main(string[] args)
         {
             MaClasse m1 = new MaClasse("toto");
             MaClasse m2 = new MaClasse("tata");
             m1.Affiche();
             m2.Affiche();
         }
        }
}
```

4.8. Passage de paramètres aux méthodes

4.8.1. Passage par référence (rappel)

Par défaut, les paramètres sont *passés par copie* (comme en C/C++). Celà signifie que le paramètre formel est une variable locale à la méthode, initialisée lors de l'appel par le paramètre effectif. Toute manipulation du paramètre formel n'a pas d'incidence sur le paramètre effectif.

Pour pouvoir modifier les paramètres au sein de la méthode, il faut ajouter le mot clé **`ref`** à la fois dans la définition de la méthode et lors de l'appel

```
   class Programme
    {
           static public void f(ref int p)
           {
                     p++;       //modifie p le paramètre effectif
           }

           static void Main()
           {
                     int a=3;
                     f(ref a);
                     Console.WriteLine(a);           // a vaut 4
           }
       }
```

Il est possible d'avoir des paramètres de *sortie seulement*.

```
    class Programme
    {
           static public void f(out int p)
           {
                     p=0;       //modifie p, sa valeur initiale non utilisée
           }

           static void Main()
           {
                     int a=3;
                     f(out a);
                     Console.WriteLine(a);           // a vaut 0
           }
       }
```

4.8.2. Méthodes avec un nombre variable de paramètres

L'argument est un tableau contenant les paramètres effectifs.

Le mot clé **`params`** indique que la fonction peut avoir un nombre variable de paramètres.

```
        class Programme
        {
```

```
            static void Fonction(params int[] p)
            {
                Console.WriteLine("{0} paramètres", p.Length);
                for(int i=0;i<p.Length;i++)
                {
                    Console.WriteLine("[{0}]",p[i]);
                }
            }

            static void Main(string[] args)
            {
                        Fonction(3);
                        Fonction(2, 5, 7);
            }
        }
```

```
Sortie Console
1 paramètres
[3]
3 paramètres
[2]
[5]
[7]
Appuyez sur une touche
pour continuer...
```

4.9. Distinction des attributs de type valeur et de type référence

Puisque l'on distingue les objets alloués dans le tas et ceux alloués dans la pile, leur traitement en tant que membres de classe diffère également.

```
using System;
namespace SyntaxeCSharp
{
    class Programme
    {
        class A
        {
           public int _valeur;  // type valeur
           public A(int valeur) {
                     _valeur = valeur;
           }
           public int GetValeur() {
                     return _valeur;
           }
        }

        class B
        {
           private double _d;   // type valeur
           private A _a;        // type référence
           public B()
           {
                 _d = 12.5;
                 _a = new A(13); // instanciation
           }
        }
        static void Main()
        {
            A objA = new A(13);
            B objB = new B();
        }
    }
}
```

4.10. Propriétés

En C#, on peut définir des *propriétés*. Il s'agit de membres qui donnent l'impression d'être des champs de données publics, alors que leur accès est contrôlé par des fonctions en lecture et en écriture.

Dans les propriétés, le mot clé **`value`** désigne la valeur reçue en écriture (bloc **`set`**).

```
using System;
namespace SyntaxeCSharp
```

```
{
    class Programme
    {
       class MaClasse
        {
            private int _valeur=2; // membre privé _valeur
            public int Valeur      // Propriété Valeur
            {
                 get {
                      return _valeur;
                 }
                 set {  // value représente la valeur int reçue en écriture
                      if(value >=0) _valeur = value;
                      else _valeur=0;
                 }
            }
        }
        static void Main()
        {
            MaClasse m = new MaClasse();
            m.Valeur = 13;    // accès en écriture -> set
            m.Valeur = -2;
            Console.WriteLine(m.Valeur);    // accès en lecture -> get
        }
    }
}
```

Les propriétés peuvent être en lecture seule. On ne fournit alors que la partie **`get`** de la propriété

```
using System;
namespace SyntaxeCSharp
{
     class MaClasse
     {
         private int _valeur=2;
         public int Valeur // Propriété Valeur en lecture seule
         {
             get {   return _valeur; }
         }
     }
     class Programme
     {
         MaClasse m=new MaClasse();
         Console.WriteLine(m.Valeur);
     }
}
```

4.11. Indexeur

Correspond à l'opérateur d'indexation du C++. Ce mécanisme permet de traiter un objet comme un tableau. Pour utiliser ce mécanisme, il faut définir une méthode **`this[ ]`** dont le paramètre est la variable qui sert d'indice.

Par exemple, l'indexeur ci-dessous utilise un indice entier et retourne un entier. La partie get gère l'accès en lecture et la partie set l'accès en écriture.

```
class MaClasse
{
    int this[int idx]
    {
       get{  ...    }
       set{  ...    }
    }
}
```

Ci-dessous un exemple où il ya deux indexeurs pour la classe **`Tableau`**.

```
using System;
using System.Collections;
namespace SyntaxeCSharp
{
       class Tableau
       {
            private ArrayList _tab;
            public Tableau(){     _tab = new ArrayList(); }

            public string this[int idx]
            {
                get{  return (string)_tab[idx];  }
                set
                 {
                      if (idx < _tab.Count) _tab[idx] = value;
                      else
                      {
                        while (idx >= _tab.Count)  _tab.Add(value);
                      }
                 }
            }

            public int this[string str]
            {
                get
                {
                    for (int idx = 0; idx < _tab.Count; idx++)
                    {
                        if ((string)_tab[idx] == str) return idx;
                    }
                    return -1;
                }
            }
       }
           class Programme
           {
                 static void Main()
                 {
                      Tableau T1 = new Tableau();
                     T1[2] = "abc";
                     T1[1] = "def";
                     Console.WriteLine(T1[0]);
                     Console.WriteLine("Indice de \"def\" : {0}", T1["def"]);
                 }
            }
}
```

4.12. Définition des opérateurs en C#

Opérateurs susceptibles d'être redéfinis :
Opérateurs unaires (1 opérande) : **`+ - ! ~ ++ -`**
Opérateurs binaires (2 opérandes) : **`+ - * / % & | ^ << >>`**

Règles d'écriture : les opérateurs sont des fonctions membres statiques **`operator__( ...)`**

Opérateur unaire : fonction membre statique avec un seul argument, du type de la classe. Doit retourner un objet, du type de la classe contenant le résultat.

Opérateur binaire : fonction membre statique avec deux arguments, du type de la classe. Doit retourner un objet, du type de la classe contenant le résultat.

```
using System;
using System.Collections;
namespace SyntaxeCSharp
{
  class MaClasse
  {
     protected int _valeur=2;
     static public MaClasse operator +(MaClasse m1, MaClasse m2)
     {
            return new MaClasse(m1._valeur + m2._valeur);
     }

     public override string ToString()
     {
            return _valeur.ToString();
     }
  }
  class Programme
  {
       static void Main()
       {
            MaClasse m1 = new MaClasse(12);
            MaClasse m2 = new MaClasse(17);
            Console.WriteLine(m1 + m2);
       }
  }
}
```

Il est possible de définir un opérateur réalisant la conversion de type.

```
using System;
namespace SyntaxeCSharp
{
   class MaClasse
   {
          private int _valeur;
          public MaClasse(int v) {
                _valeur=v;
          }
          public static explicit operator int(MaClasse m) {
             return m._valeur;
          }
   }
    class Programme
    {
         static void Main(){
             MaClasse m1=new MaClasse();
             int i;
             i=(int)m1; // Conversion explicite de MaClasse -> int
          }
    }
}
```

4.13. Conversion Numérique ↔ Chaîne (format/parse)

4.13.1. Formatage (conversion numérique vers string)

Tout objet dérive de **`object`** et dispose donc d'une méthode **`ToString()`.**

```
          int d=12;
          string s=d.ToString();
          Console.WriteLine(s);
```

Remarque : il y a aussi pour les types numériques une méthode **`ToString(string format, IFormatProvider provider)`** qui permet de préciser le formattage (cf section sur la classe string)

4.13.2. Parsing (conversion string vers numérique)

Les types numériques (int,float,...) disposent d'une méthode **`Parse()`**. Cette méthode génère une exception si le format de la chaîne est incorrect

`static T  Parse(string str)` [T est le type numérique]

```
string s;
s=Console.ReadLine(); //lit une chaîne
int var;
try
{
          var = int.Parse(s);
          var = int.Parse("13");
}
catch (Exception e)
{
  Console.WriteLine(e.Message);
}
```

Une seconde méthode **`TryParse()`**, plus rapide, retourne un booléen pour indiquer le succès ou l'échec de la conversion. Cette méthode ne déclenche pas d'exception.

`static bool TryParse(string str, out T var)`

```
string s;
s=Console.ReadLine(); //lit une chaîne
int var;
if(int.TryParse(s,out var)==false)
{
          Console.WriteLine("La chaîne ne respecte pas le format int");
}
```

4.14. Les énumérations (types valeur)

Un type créé avec le mot clé **`enum`** représente des variables pouvant prendre un nombre fini de valeurs prédéfinies. Une variable d'un type **énumération** peut prendre pour valeur une des valeurs définies dans la liste décrivant l'énumération :

```
enum Feu{vert,orange,rouge}
...
Feu f;       // f est une variable
f=Feu.vert; //Feu.vert est une valeur
```

Les valeurs d'une énumération correspondent à des valeurs entières qu'on peut choisir. On peut choisir aussi le type entier sous-jacent. On peut transtyper une valeur énumération en valeur entière. On peut aussi appliquer certaines opérateurs :

`== (compare), +, -, ++, - , &, |,~`

```
using System;
namespace SyntaxeCSharp
{
    enum Color { red, blue, green };      // vaut 0,1,2
    enum Access { personal = 1, group = 2, all = 4 };
    enum AccessB : byte { personal = 1, group = 2, all = 4 };  // codé sur 8 bits
```

```
    class Programme
    {
        static void Main()
        {
            Color c = Color.blue;
            Console.WriteLine("Couleur {0}",(int)c); // Couleur 1
            Access a1 = Access.group;
            Access a2 = Access.personal;
            Console.WriteLine((int)a1);
            Console.WriteLine((int)a2);
            Access a3 = a1 | a2;          // ou bit à bit sur entier sous-jacent
            Console.WriteLine((int)a3);
        }
    }
}
```

4.15. Les structures

Le mot-clé **`struct`** permet de définir des *types valeur* ayant des méthodes et des atttributs. L'allocation d'une variable de type valeur se fait sur la pile alors que les objets de type référence sont alloués dans le tas. L'intérêt des types struct (type valeur) est de pouvoir gérer les objets plus efficacement (accès mémoire plus rapide et ne nécessite pas le garbage collector). Certaints types fournis par .NET sont des types struct. En contrepartie, certaines restrictions s'appliquent aux types créés avec le mot clé `struct`.

Particularités des types créés avec **`struct`** :

- Les structures peuvent être converties en type **`object`** (comme tout type)
- Les structures ne peuvent dériver d'aucune classe ou structure.
- Les structures ne peuvent pas servir de base pour la dérivation.
- Un type structure peut contenir un/des constructeur(s) mais *pas de constructeur sans argument*
- Les champs ne peuvent pas être initialisés en dehors des constructeurs
- Un type structure peut implémenter une interface

Il y a toujours un constructeur par défaut (sans argument) qui fait une initialisation par défaut (0 pour les valeurs numériques). Ce constructeur reste utilisable même si un autre constructeur a été défini. On peut définir des propriétés, des indexeurs dans les structures.

Les structures doivent être utilisées principalement pour les objets transportant peu de données membre. Au delà de quelques octets, le bénéfice obtenu du fait que l'allocation se fait dans la pile est perdu lors des passages de paramètres, puisqu'un objet de type valeur est dupliqué lorsque l'on fait un passage par valeur.

La portée d'une variable de type valeur va de la déclaration jusqu'à la fin du bloc de la déclaration (puisque les variables sont alors stockées sur la pile).

```
using System;
namespace SyntaxeCSharp
{
    struct MaStructure
    {
        private int _valInt;              // pas d'initialisation ici
        private double _valDouble;
        public MaStructure(int vI, double vD){
            _valInt = vI;
            _valDouble = vD;
        }
        public int ValInt       // propriété ValInt
        {
            get { return _valInt; }
        }
```

```
        public double ValDouble // propriété ValDouble
        {
            get { return _valDouble; }
            set { _valDouble = value; }
        }
    } // MaStructure

    class Programme
    {
        static void Main()
        {
           // même si on utilise new → stocké dans la pile
            MaStructure s = new MaStructure(12,3.4);
            MaStructure s2 = new MaStructure();
            Console.WriteLine(s2.ValInt);
        } // fin de portée de s1 et s2
    }
}
```

Le framework .NET fournit quelques types valeurs définis comme des structures. Pour la programmation Windows Forms, on a souvent recours aux structures ci-dessous

`System.Drawing.Point` : décrit un point du plan à coordonnées entières int.
`System.Drawing.Rectangle` : rectangle
`System.Drawing.Size` : taille d'une fenêtre

Les structures `System.Drawing.PointF/System.Drawing.RectangleF/System.Drawing.SizeF` correspondent à la version avec attributs en type flottant.

4.16. Notation UML des classes.

UML (Unified Modelling Language) est une représentation graphique pour les objets. L'UML comprend différentes représentation graphiques. Parmi les différents diagrammes, le diagramme de classes représente la structure des classes (attributs et méthodes) ainsi que les relations entre les classes (et donc entre les objets).

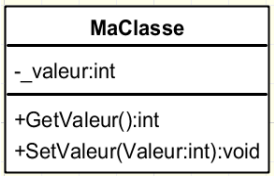
Chaque classe est représentée par un rectangle.

La première partie décrit les attributs, la seconde les méthodes. La visibilité est décrite par un symbole

- privé

+ public

protégé



Le diagramme précédent représente la classe C# ci-dessous. Notons que le détail des méthodes n'est pas indiqué dans ce diagramme. Seul le nom des méthodes évoque le traitement sous-jacent.

```
class MaClasse
{
          private int _valeur;
          public int GetValeur() { ... }
          public void SetValeur(int Valeur) { ….  }
}
```

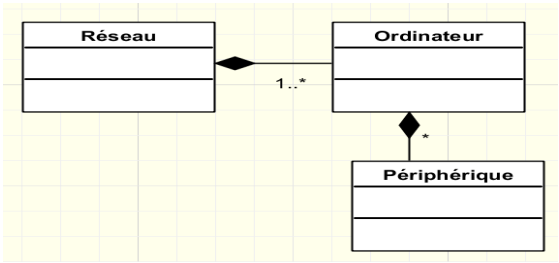
5. La composition (objet avec des objets membres)

La programmation orientée objet permet de décrire les applications comme des collaborations d'objets. Par conséquent, dans les diagrammes de classe, certaines relations apparaissent entre les classes. Par exemple, certaines méthodes d'une classe utilisent les services d'une autre classe. Il n'est pas question de rentrer dans les détails de la modélisation par UML, mais on peut toutefois s'intéresser à certaines relations importantes comme

la composition et la dérivation (cf. Section suivante)

La composition décrit la situation où un objet est lui même constitué d'objets d'autres classes. Par exemple un ordinateur est composé d'une unité centrale et de périphériques. Un livre est constitué de différents chapitres.

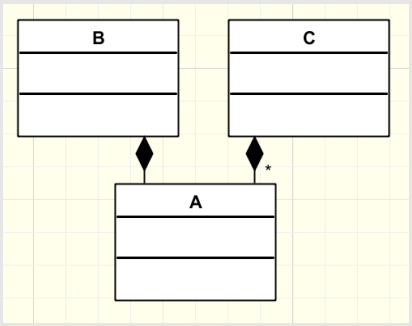
Cette relation particulière est décrite par une relation où un losange est dessiné côté composite (l'objet qui agrège).



Pour le diagramme ci-contre, la relation entre la classe Réseau et la classe Ordinateur signifie : "un réseau est constitué de 1 ou plusieurs ordinateurs (1..*) "

Dans le langage objet cible, cette relation de composition est simplement décrite par le fait qu'une classe puisse avoir un ou plusieurs membres de type classe (ou struct).

```
class A{    }

   class B
   {
       private A _objet;

       public B()
       {
           _objet = new A();
       }
   }

   class C
   {
       private A[] _objets;

       public C(int nb)
       {
           _objets = new A[nb];

       }
   }
```



Les classes ci-dessus ont les relations suivantes :

un objet de la classe B est constitué d'un objet de la classe A

un objet de la classe C est constitué de 0, 1 ou plusieurs objets de la classe A puisque un objet de type C stocke éventuellement plusieurs objets de type A.

6. La dérivation

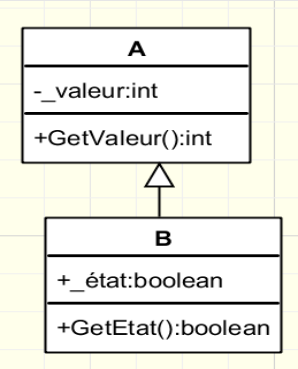
La relation de composition (un objet est composé d'autres objets) n'est pas la seule relation entre les objets. La dérivation est une seconde relation essentielle de la programmation orientée objet.

La dérivation est un mécanisme fourni par la langage qui permet de spécialiser (la classe décrit des objets plus spécifiques) une classe existante en lui ajoutant éventuellement de nouveaux attributs / de nouvelles méthodes. La classe dérivée possède alors intégralement les attributs et les méthodes de la classe de base. On peut donc considérer un objet de la classe dérivée comme un objet de la classe de base, avec des membres en plus. C'est une façon de ré-utiliser une classe existante.

Ci-dessous, la classe B dérive de la classe A (c'est indiqué dans l'entête de la classe B). La classe B possède donc automatiquement les attributs et les méthodes de la classe A, avec en plus des membres spécifiques.

On dit aussi que la classe B hérite des attributs et des méthodes de sa super-classe.

```
class Program
   {
       class A
       {
           private int _valeur;

           public A(int val)
           {
               _valeur = val;
           }

           public int GetValeur() { return _valeur; }
       }

       class B :  A
       {
           private bool _etat;

           public B(int valeur, bool etat):base(valeur)
           {
               _etat = etat;
           }

           public bool GetEtat()
           {
               return _etat;
           }
       }

       static void Main(string[] args)
       {

           A objA = new A(12);
           Console.WriteLine(objA.GetValeur());

           B objB = new B(17, true);
           Console.WriteLine(objB.GetValeur());
           Console.WriteLine(objB.GetEtat());
       }
   }
```



En notation UML, la relation de dérivation est notée avec une flèche partant de la classe dérivée (ou sous-classe) vers la classe de base.

Un objet de la classe A possède un attribut et une méthode.

Un objet de la classe B possède deux méthodes (GetEtat() et GetValeur()) et deux attributs (_etat et _valeur).

La classe B dérive de la classe A.

B est une sous-classe de la classe A.

A est super-classe ou classe parent de la classe B.

Important : sur le plan de la compatibilité de type, un objet de la classe B peut toujours être considéré comme un objet de la classe A aussi (puisqu'il possède au moins les attributs et méthodes de la classe de base).

6.1. Syntaxe

On peut définir une classe comme sous-classe d'une classe existante de la façon suivante : la classe Derivee est sous-classe de la classe MaClasse. Ce qui signifie que tout objet de la classe Derivee dispose des attributs et méthodes de sa (ses) super-classe(s).

```
class Derivee : MaClasse
    {
            protected int _increment;
            public Derivee(int val): base(12)
            {
                 _increment=val;
            }
    }
```

Tout objet de la classe dérivée dispose des attributs et méthodes hérités de la classe de base (MaClasse) ainsi que des attributs et méthodes spécifiques décrits dans cette classe (partie incrémentale).

On peut également spécifier qu'une classe ne puisse être dérivée (sealed class).

```
   sealed class MaClasse
   {
      // cette classe ne peut pas être dérivée
   }
```

6.2. Différence de visibilité protected/private

Un membre privé n'est pas accessible par l'utilisateur de la classe, ni par un objet de la classe dérivée.

Un membre protected n'est pas accessible par l'utilisateur de la classe mais est accessible par un objet de la classe dérivée. Voir ci-dessous le membre _valeur.

6.3. Initialisateur de constructeur (dans le cas de la dérivation)

Le mot clé **`base( )`** utilisé au niveau du constructeur indique quel constructeur de la classe de base utiliser pour initialiser la partie héritée d'un objet **`Derivee`**.

```
using System;
namespace SyntaxeCSharp
{
        class MaClasse{
                protected int _valeur;
                public MaClasse(int v){  _valeur=v;   }
        }// MaClasse

        class Derivee : MaClasse
        {
            protected double _increment;

            public Derivee(double valInc,int valP): base(valP){
                        _increment=valInc;
            }
             public void Set(double inc, int val)
             {
                 _valeur = val;        // _valeur accessible car protected
                 _increment = inc;

             }
        } // Derivee

    class Programme
    {
```

```
        static void Main()
        {
            MaClasse m1=new MaClasse(2);

            Derivee d1=new Derivee(4.3,12);

            d1.Set(2.3, 2);
        }
    }
}
```

6.4. Par défaut, les méthodes ne sont pas virtuelles (ligature statique)

De même qu'en C++, les méthodes sont non virtuelles par défaut. Ce qui signifie qu'à partir d'une référence de type MaClasse, on ne peut appeler que des méthodes de cette classe ou de ses classes de base. Y compris si la référence désigne en réalité un objet d'une classe dérivée de MaClasse, ce qui est possible avec la compatibilité de type. On appelle ça une ligature statique des méthodes. La méthode invoquée est alors désignée à la compilation.

Dans le cas de la surdéfinition d'une méthode (non virtuelle) de la classe de base dans la classe dérivée, il faut tout de même préciser qu'on surdéfinit la méthode (on reconnaît qu'il y a déja une méthode de même nom dans l'ascendance) en utilisant le mot-clé **`new.`**

```
using System;
namespace SyntaxeCSharp
{
   class MaClasse
    {
        public void Methode(){
            Console.WriteLine("MaClasse.Methode");
        }
    }
   class Derivee : MaClasse
    {
        new public void Methode(){
              // masque celle de la classe de base
            Console.WriteLine("Derivee.Methode");
        }
    }
   class Programme
    {
        static void Main()
        {

              MaClasse m=new MaClasse();
              Derivee d=new Derivee();
              m.Methode();
              d.Methode();
              m=d;
              m.Methode();
        }
    }
}
```

```
Sortie Console
MaClasse.Methode
Derivee.Methode
MaClasse.Methode
Appuyez sur une touche pour
continuer...
```

6.5. Définition des méthodes virtuelles (méthodes à ligature dynamique)

Le mot clé **`virtual`** doit être ajouté pour que l'appel d'une méthode bénéficie d'une recherche *dynamique* de la "bonne" méthode à l'exécution. En outre, la surdéfinition d'une méthode virtuelle dans la classe dérivée doit être précédée du mot-clé **`override`** (on reconnaît ainsi qu'il y a déja une méthode de même nom dans l'ascendance et que celle-ci est virtuelle)

```
using System;
namespace SyntaxeCSharp
{
```

```
   class MaClasse
    {
        virtual public void Methode(){
            Console.WriteLine("MaClasse.Methode");
        }
    }
   class Derivee : MaClasse
    {
        override public void Methode(){
            Console.WriteLine("Derivee.Methode");
        }
    }
   class Programme
    {
        static void Main()
        {

              MaClasse m=new MaClasse();
              Derivee d=new Derivee();
              m.Methode();
              d.Methode();
              m=d;
              m.Methode();
        }
    }
}
```

Sortie Console
```
MaClasse.Methode
Derivee.Methode
Derivee.Methode
Appuyez sur une touche pour
continuer...
```

6.6. Une propriété peut être surdéfinie dans la classe dérivée

Qu'elle soit virtuelle ou non, une propriété peut être surdéfinie dans la classe dérivée. Les mêmes règles s'appliquent que pour les autres méthodes.

```
using System;

namespace SyntaxeCSharp
{
    class Personne
    {
        protected string _nom;
        protected string _prenom;
        public Personne(string nom, string prenom){
             _nom = nom;
             _prenom = prenom;
        }
        public string Identite{
             get { return (_prenom + " " + _nom); }
        }
    } // Personne

    class Enseignant : Personne
    {
        private int _section;
        public Enseignant(string nom, string prenom, int section):base(nom,prenom){
             _section = 27;
        }
        public new string Identite      // redéfinition
        {
             get{       return String.Format("{0} section {1}",
                                                 base.Identite,
                                                 _section);
             } //fin get
        } // fin propriété
    } // Enseignant
```

```
    class Programme
    {
        static void Main()
        {
            Enseignant e=new Enseignant("Saubion","Frédéric",27);
            Console.WriteLine(e.Identite);
            Personne p = e;
            Console.WriteLine(p.Identite);
        }
    }
}
```

```
Sortie Console
Saubion Frédéric section 27
Saubion Frédéric
Appuyez sur une touche pour
continuer...
```

```
using System;
namespace SyntaxeCSharp
{
   class Personne
   {
        ...
        virtual public string Identite{
            get { return (_prenom + " " + _nom); }
        }
   } // Personne
   class Enseignant : Personne
   {
        ...
        override public string Identite
        {
            get{       return String.Format("{0} section {1}",
                                            base.Identite,
                                            _section);
            }
        }
   }//Enseignant

    class Programme
    {
        static void Main()
        {
            Enseignant e=new Enseignant("Saubion","Frédéric",27);
            Console.WriteLine(e.Identite);
            Personne p = e;
            Console.WriteLine(p.Identite);     // retrouvée dynamiquement
        }
    }
}
```

```
Sortie Console
Saubion Frédéric section 27
Saubion Frédéric section 27
Appuyez sur une touche pour
continuer...
```

6.7. Classe object (System.Object)

En .NET, le type **object** (System.Object) est classe racine de tout type. Tout peut donc être transformé en type **object**. Au besoin, pour les types valeur, l'opération de boxing permet de créer un objet compatible avec **object**. La classe object dispose des méthodes ci-dessous

bool Equals(object o) : (virtuelle) indique l'égalité d'objets. Pour les types valeurs, l'égalité de valeur. Pour les types référence, l'égalité de référence. Attention, ce mécanisme peut être dynamiquement substitué. On doit donc consulter la documentation du type pour connaître le rôle exact de cette méthode.

static bool Equals(object objA,object objB) : idem en version statique

Type GetType(): retourne un descripteur de type

static bool ReferenceEquals(object objA, object objB): teste si les deux références désignent le même objet. Cette méthode est non substituable. Elle réalise donc toujours ce traitement.

virtual string ToString(): retourne le nom du type sous forme de chaîne. Peut être substitué dans les classes dérivées pour décrire un objet sous forme d'un chaîne.

```
static void Main(string[] args)
{
  int i = 1;
  MaClasse m = new MaClasse();

  object Obj = i;
  Console.WriteLine(Obj.GetType().FullName);
  Console.WriteLine(Obj.ToString());

  Obj = m;
  Console.WriteLine(Obj.GetType().FullName);
  Console.WriteLine(Obj.ToString());
}
```

Sortie

```
System.Int32
1
SyntaxeCSharp.MaClasse
SyntaxeCSharp.MaClasse
Appuyez sur une touche pour
continuer...
```

Classe **System.Collections.ArrayList** : stocke des références à des variables **object.** Comme object est l'ancêtre (ou classe de base) de tout type .NET, tout objet (ou variable) peut être considérée comme de type object. Les conteneurs ArrayList peuvent donc stocker n'importe quel élément, voire même des éléments de types différents dans le même conteneur.

```
ArrayList  t=new ArrayList();

t.Add(new MaClasse(12));
t.Add(12L);
t.Add("toto");

foreach(object obj in t)
{
   Console.WriteLine(obj.GetType().Name);
   Console.WriteLine(obj);
}
```

6.8. Boxing/unboxing

.NET fait la distinction type valeur/type référence pour que la gestion des variables avec peu de données soit plus efficace. Pour conserver une uniformité d'utilisation de tous les types, même les variables de type valeur peuvent être considérées comme de type référence. L'opération qui consiste à créer un objet de type référence à partir d'une variable de type valeur est appelée **Boxing.** L'opération inverse permettant de retrouver une variable à partir d'un type référence est appelée **Unboxing**.

Boxing : conversion type valeur vers type référence
Unboxing : conversion inverse

Grâce au boxing, on peut considérer une variable int (qui est pourtant un type primitif) en objet compatible avec les autres classes .NET. Le boxing consiste à créer une variable sur le tas qui stocke la valeur et à obtenir une référence (compatible object) sur cet objet. L'opération de boxing a donc un coût à l'exécution. Mais ce mécanisme garantit que toute variable/objet puisse être compatible avec le type ancêtre **object**.

```
class Programme
{
        static void Main()
        {
            int var = 2;
            object obj = var;    // Boxing
            Console.WriteLine(obj.GetType().FullName);
             int var2 = (int)obj; //Unboxing
        }
}
```

6.9. Reconnaissance de type (runtime)

Avec .NET (pas seulement C#), les types ont une description stockée dans les modules compilés. Le compilateur n'est pas le seul a connaître les types utilisés. Ces types sont décrits par des méta-données stockées dans les modules compilés. On peut donc connaître à l'exécution (runtime) beaucoup d'informations sur les objets manipulés et leurs types. Tous les types sont représentés par un descripteur de type de la classe **`System.Type`** qui décrit exhaustivement les caractéristiques du type : nom, taille, méthodes et signatures …

Ce mécanisme est dans l'interface de la classe de base `object`. Tout objet (et même toute variable) dispose donc d'une méthode **`GetType`** qui fournit un descripteur du type de l'objet.

```
namespace ExempleType
{
    class Personne
    {
        protected string _nom;
        protected string _prenom;
        public Personne(string nom, string prenom)
        {
            _nom = nom;
            _prenom = prenom;
        }
        public string Identite
        {
            get { return (_prenom + " " + _nom); }
        }
    } // Personne

    class Program
    {
        static void Main(string[] args)
        {
            Personne p = new Personne("Durand", "Jacques");
            Console.WriteLine(p.Identite);

            Type t = p.GetType();

            Console.WriteLine(t.FullName);
            Console.WriteLine("Liste des méthodes de ce type:");
            System.Reflection.MethodInfo[] liste = t.GetMethods();
            foreach(System.Reflection.MethodInfo minfo in liste)
            {
                Console.WriteLine(minfo.Name);
            }
        }
    }
}
```

Sortie

```
Jacques Durand
ExempleType.Personne
Liste des méthodes de ce
type:
get_Identite
ToString
Equals
GetHashCode
GetType
Appuyez sur une touche pour
continuer...
```

6.10. Exemple de synthèse sur la dérivation

```
using System;
namespace SyntaxeCSharp
{
    class Base
    {
     private    int _prive;
     protected  double _protege;
     public Base(int vI, double vD)
     {
         _prive = vI;
         _protege = vD;
     }
```

```
     public void A()         { Console.WriteLine("Base.A()");}
     virtual public void B() { Console.WriteLine("Base.B()");}
     public void C()         { Console.WriteLine("Base.C()");}
     } //Base


    class Derivee : Base
    {
        protected string _str;

        public Derivee(string vS, int vI, double vD): base(vI, vD)
        {
                _str = vS;
                // _prive n'est pas accessible ici
                // _protege est accessible
        }

          new public void A()       { Console.WriteLine("Derivee.A()");}
          override public void B()  { Console.WriteLine("Derivee.B()");}
          public void D()           { Console.WriteLine("Derivee.D()"); }
    } // Derivee


    class Programme
    {
        static void Main()
        {
           Derivee d = new Derivee("abc",13, 4.5);
            d.A();
            d.B();
            d.C();
            d.D();
            Base bD = d;
            bD.A();
            bD.B();
            bD.C();
        }
    }
}
```

```
Sortie Console
Derivee.A()
Derivee.B()
Base.C()
Derivee.D()
Base.A()
Derivee.B()
Base.C()
Appuyez sur une touche pour
continuer...
```

6.11. Conversions de type

6.11.1. Notation des casts

Utilise la notation de cast du C/C++. La conversion Dérivé vers Base est toujours possible. On peut par exemple toujours stocker une référence à n'importe quel objet dans une référence de type **`object`**. Dans l'autre sens, si la conversion est possible, elle nécessite un cast.

```
          class Derivee:Base {  }
          ...

          Base b=new Derivee();  //OK
          Derivee d=(Derivee)b;  // si b est de type Derivee OK
                    // sinon, déclenche exception System.InvalidCastException
```

6.11.2. Opérateur as

```
          Base b;
          ...
          Derivee d=b as Derivee;
                    // si b ne désigne pas un objet Derivee alors d==null
```

6.11.3. Opérateur is

```
Base b;
...
if(b is Derivee)  // teste si l'objet désigné par b est de type compatible
{
          Derivee d =(Derivee)b; // pas d'exception
          ...
}
```

6.11.4. Typeof/Sizeof

Le mot-clé **typeof** retourne le descripteur de type pour un type donné.

```
Type t =typeof(int);
```

L'opérateur **sizeof** ne s'applique ***qu'à un type valeur*** et retourne la taille d'une variable de ce type

```
Console.WriteLine(sizeof(int));
```

7. Interfaces et classes abstraites

7.1. Interfaces

Une ***interface*** est un ensemble de prototypes de méthodes, de propriétés ou d'indexeurs qui forme un ***contrat***. Une classe qui décide d'implémenter une interface s'engage à fournir une implémentation de toutes les méthodes prévues par l'interface.

En contrepartie, dès lors qu'une classe implémente une interface donnée, elle contient nécessairement un comportement minimal, celui prévu par l'interface.

Une interface décrit uniquement un ensemble de prototypes de méthodes que devront implémenter les classes qui veulent respecter cette interface :

- pas de qualificatif public/private/protected pour les membres
- pas de données membres
- pas de qualificatif virtual

Ci-dessous une interface avec 2 méthodes et deux classes implémentant cette interface.

```
using System;
namespace SyntaxeCSharp
{
    interface IMonBesoin
    {
        void A();
        void B();
    }

      class MaClasse:IMonBesoin
      {
        public void A(){ Console.WriteLine("MaClasse.A()");}
        public void B() { Console.WriteLine("MaClasse.B()"); }
      } // MaClasse

      class AutreClasse : IMonBesoin
      {
        public void A() { Console.WriteLine("AutreClasse.A()"); }
        public void B() { Console.WriteLine("AutreClasse.B()"); }
      } // MaClasse
```

```
      class Programme
      {
          static void Main()
          {
             IMonBesoin ib=new MaClasse();
             ib.A();
             ib.B();

             ib=new AutreClasse();
             ib.A();
             ib.B();
          }
    }
}
```

```
Sortie Console

MaClasse.A()
MaClasse.B()
AutreClasse.A()
AutreClasse.B()
Appuyez sur une touche pour
continuer...
```

Un objet implémentant l'interface **IMonBesoin** peut être référencé au moyen d'une référence de type **IMonBesoin.** Cette référence ne permet d'accéder qu'aux méthodes décrites dans l'interface.

7.1.1. Les méthodes d'une interface ne sont pas virtuelles.

```
using System;
namespace SyntaxeCSharp
{
    interface IMonBesoin
    {
        void A();
        void B();
    }

    class MaClasse:IMonBesoin
    {
        public void A()         { Console.WriteLine("MaClasse.A()");}
        public void B()         { Console.WriteLine("MaClasse.B()"); }
    }

    class DeriveeMaClasse : MaClasse
    {
        new public void A() { Console.WriteLine("DeriveeMaClasse.A()"); }
        new public void B() { Console.WriteLine("DeriveeMaClasse.B()"); }
    }

    class Programme
    {
        static void Main()
        {
            IMonBesoin ib=new DeriveeMaClasse();
            ib.A();
            ib.B();
        }
    }
}
```

```
Sortie Console
MaClasse.A()
MaClasse.B()
Appuyez sur une touche pour
continuer...
```

Si l'on souhaite que les méthodes soient virtuelles, il faut le préciser dans la classe **MaClasse**.

```
using System;

namespace SyntaxeCSharp
{
    interface IMonBesoin
    {
        void A();
        void B();
    }
```

```
    class MaClasse:IMonBesoin
    {
        virtual public void A() { Console.WriteLine("MaClasse.A()");}
        virtual public void B() { Console.WriteLine("MaClasse.B()"); }
    }

    class DeriveeMaClasse : MaClasse
    {
        override public void A() { Console.WriteLine("DeriveeMaClasse.A()"); }
        override public void B() { Console.WriteLine("DeriveeMaClasse.B()"); }
    }

    class Programme
    {
        static void Main()
        {
            IMonBesoin ib=new DeriveeMaClasse();
            ib.A();
            ib.B();
        }
    }
}
```

```
Sortie Console
DeriveeMaClasse.A()
DeriveeMaClasse.B()
Appuyez sur une touche pour
continuer...
```

7.1.2. Forcer à utiliser l'abstraction

On peut forcer l'utilisateur à utiliser une classe via son abstraction définie par une interface. Ci-dessous, la méthode A de l'implémentation ne peut être appelée qu'à partir d'une référence de type **`IMonBesoin`**.

```
using System;
namespace SyntaxeCSharp
{
    interface IMonBesoin
    {
        void A();
        void B();
    }
    class MaClasse:IMonBesoin
    {
        void IMonBesoin.A()     { Console.WriteLine("MaClasse.A()");}
        public void B()         { Console.WriteLine("MaClasse.B()"); }
    }

    class Programme
    {
        static void Main()
        {
            MaClasse mc = new MaClasse();
            // mc.A();  on ne peut pas utiliser A sur objet MaClasse
            mc.B();
            IMonBesoin ib = mc;
            ib.A();     // on peut utiliser A ici
            ib.B();

        }
    }
}
```

7.1.3. Interfaces standard .NET

Le framework .NET fournit un ensemble de classes mais aussi un ensemble d'interfaces que certaines classes doivent implémenter pour bénéficier de mécanismes ad hoc.

Interface **`System.Collections.IEnumerable`** : une classe qui implémente cette interface (généralement un conteneur) fournit un moyen de parcourir ses instances. L'interface **`IEnumerable`** standardise le parcours des conteneurs .NET. En particulier, un objet qui implémente `IEnumerable` peut être parcouru au moyen de

l'instruction **foreach**. (cf. Section sur les conteneurs)
Interface **System.ICloneable** : une classe qui implémente cette interface fournit un moyen de créer une copie d'un objet de la classe (objet clone).

7.1.4. Tester qu'un objet implémente une interface (opérateurs is et as)

On peut tester, à l'exécution, si un objet donné implémente ou non une interface donnée. On utilise là encore les opérateurs **is** et **as** du langage qui s'appuient sur le CTS pour vérifier la compatibilité de type.

```
using System;
namespace SyntaxeCSharp
{
    interface IMonBesoin
    {
        void A();
        void B();
    }
    class MaClasse:IMonBesoin
    {
        public void A()         { Console.WriteLine("MaClasse.A()");}
        public void B()         { Console.WriteLine("MaClasse.B()"); }
    }

    class Programme
    {
        static void Main()
        {
            object obj = new MaClasse();

            if (obj is IMonBesoin)
            {
                Console.WriteLine("Oui, cet objet implémente l'interface IMonBesoin");
            }

            IMonBesoin ib = obj as IMonBesoin;

            if (ib == null)
            {
                Console.WriteLine("Non, obj n'implémente pas l'interface IMonBesoin");
            }
            else
            {
                Console.WriteLine("Oui, obj implémente l'interface IMonBesoin");
            }
        }
    }
}
```

7.2. Classes abstraites

Les interfaces ne peuvent décrire qu'une liste de méthodes qu'une classe devra implémenter. On ne peut pas définir de méthode dans une interface ni même déclarer des données membres. Une classe abstraite correspond aux classes abstraites du C++. Les méthodes n'ont pas toutes une implémentation. Celle qui n'ont pas d'implémentation sont marquée **abstract**. En revanche, certaines méthodes concrètes peuvent être définies au niveau d'une classe abstraite. C'est ce qui les différencie des interfaces.

Puisqu'une classe abstraite ne définit pas toutes ses méthodes, il n'est pas possible d'instancier une classe abstraite. En revanche, une classe abstraite sert de base dans le cadre de la dérivation. Toutes les méthodes et attributs définis dans une classe abstraite sont disponibles dans les classes qui en dérivent.

Ci-dessous, la classe abstraite décrit un système qu'on peut démarrer et arrêter. Tout système a un nom. Ce traitement élémentaire est décrit directement dans la classe abstraite. Toute classe dérivant de SystemAbstrait

hérite de ce nom et de son accesseur. Toutes les méthodes **abstraites** sont **virtuelles**.

```
using System;

namespace SyntaxeCSharp
{
    abstract class SystemeAbstrait
    {
        private string _nom;
        public SystemeAbstrait(string Nom)
        {
            _nom=Nom;
        }
        public string GetNom()  // méthode concrète
        {
            return _nom;
        }
        abstract public void Start();  // méthode abstraite
        abstract public void Stop();
        abstract public bool IsStarted();
    }

    class SystemV1 : SystemeAbstrait
    {
        private bool _on;
        public SystemV1(string Nom)
            : base(Nom)
        {
            _on = false;
        }
        override public void Start()  // fournit une implémentation
        {
            _on = true;
        }
        override public void Stop()
        {
            _on = false;
        }

        override public bool IsStarted()
        {
            return _on;
        }
    }

    class Programme
    {
        static void Main()
        {
           SystemeAbstrait sys = new SystemV1("Version 1");
           Console.WriteLine(sys.GetNom());
           sys.Start();
        }
    }
}
```

8. Exceptions

De nombreuses fonctions C# sont susceptibles de générer des exceptions, c'est à dire des erreurs. Une exception est déclenchée lorsqu'un comportement anormal se produit. Lorsqu'une fonction est susceptible de générer une exception, le programmeur devrait la gérer dans le but d'obtenir des programmes plus résistants aux erreurs : il faut toujours éviter le "plantage" sauvage d'une application. La gestion d'une exception se fait selon le schéma suivant :

```
try{
        code susceptible de générer une exception
}
catch (Exception e){
        traiter l'exception e
}
instruction suivante
```

Si la fonction ne génère pas d'exception, on passe alors à `instruction suivante`, sinon on passe dans le corps de la clause **`catch`** puis à `instruction suivante`. Notons les points suivants :

• `e` est un objet de type **`Exception`** ***ou dérivé***. On peut être plus précis en utilisant des types tels que `IndexOutOfRangeException`, `FormatException`, `SystemException`, etc… : il existe plusieurs types d'exceptions.

En écrivant catch (`Exception` e), on indique qu'on veut gérer tous les types d'exceptions. Si le code de la clause `try` est susceptible de générer plusieurs types d'exceptions, on peut vouloir être plus précis en gérant l'exception avec plusieurs clauses catch :

```
try{
        code susceptible de générer des exceptions
}
catch (IndexOutOfRangeException e1)
{
        traiter l'exception e1
}
catch (FormatException e2)
{
        traiter l'exception e2
}
instruction suivante
```

• On peut ajouter aux clauses try/catch, une clause **`finally`** :

```
try{
        code susceptible de générer une exception
}
catch (Exception e)
{
        traiter l'exception e
}
finally
{
        code toujours exécuté (après try ou catch selon le cas)
}
instruction suivante
```

Qu'il y ait exception ou pas, le code de la clause `finally` sera toujours exécuté. Dans la clause `catch`, on peut ne pas vouloir utiliser l'objet `Exception` disponible. Au lieu d'écrire catch (`Exception` e){..}, on écrit alors `catch(Exception){...}` ou plus simplement `catch {...}`.

La classe **`Exception`** a une propriété **`Message`** qui est un message détaillant l'erreur qui s'est produite.

La classe `Exception` a aussi une méthode **`ToString`** qui rend une chaîne de caractères indiquant le type de l'exception ainsi que la valeur de la propriété **`Message`**. On pourra ainsi écrire :

```
catch (Exception ex)
{
        Console.WriteLine("L'erreur suivante s'est produite : {0}",
                          ex.ToString());
```

```
                ...
        }//catch
```

On peut écrire aussi :

```
        catch (Exception ex)
        {
                Console.WriteLine("L'erreur suivante s'est produite : {0}",ex);
                ...
        }//catch
```

Le compilateur va attribuer au paramètre {0}, la valeur *ex.ToString()*.

Un exemple où l'on exploite les exceptions liées aux dépassements de valeur numérique.

```
L'opération arithmétique a provoqué un dépassement de capacité.

100

Appuyez sur une touche pour continuer...
```

```
class Program
    {
        static void Main(string[] args)
        {
            sbyte octet = 100;

            // bloc où  des exceptions sont éventuellement déclenchées
            try
            {
                checked  // indique que les dépassements déclenchent des exceptions
                {
                    octet += octet;
                }

            }
            catch (Exception e)  // intercepte les exceptions
                                 // de type Exception (ou compatible)
            {
                Console.WriteLine(e.Message); // Message lié à l'exception
            }
            finally  // Exécuté après try{} si pas d'exception
                     // ou après catch{ } si exception
            {
                Console.WriteLine(octet.ToString());  // affiche la valeur de octet
            }
        }
    }
```

On peut également préciser le type d'exception, ici il s'agit de **`OverflowException`**

```
class Program
    {
        static void Main(string[] args)
        {
            sbyte octet = 100;

            // bloc où  des exceptions sont éventuellement déclenchées
            try
            {
                checked
                {
                    octet += octet;
                }
            }
```

```
            catch (System.OverflowException e)
                    // intercepte les exceptions de type OverflowException
            {
                Console.WriteLine(e.Message);
            }
        }
    }
```

9. Divers

9.1. Types imbriqués

On peut créer un type à l'intérieur d'un autre type. Ci-dessous, la classe **B** est déclarée à l'intérieur de la classe **A**. Les méthodes de la classe **B** ont donc accès à la partie privée d'un objet de la classe A. Puisque le type B est public, on peut aussi instancier un objet de la classe **A.B** (la classe B définie dans A).

```
using System;
namespace SyntaxeCSharp
{
    class A
    {
       public class B
       {
            private A _refA;
            public B()         _refA = null; }
            public B(A rA){    _refA = rA;}

            public void ResetA(){
                if(_refA!=null) _refA.x = 0;
            }
       }

       private int x = 2;
       private B _refB;

       public A(){    _refB = new B(this);        }
       public void Reset(){   _refB.ResetA();     }
    }

    class Programme
    {
        static void Main()
        {
            A a = new A();
            a.Reset();
            A.B b = new A.B();
        }
    }
}
```

9.2. Dérivation et vérification de type

Le fichier ci-dessous décrit trois classes en relation hiérarchique de dérivation. La classe C dérive de B qui elle-même dérive de A.

Comme l'on peut désigner tout objet par une référence de type object, il est important de savoir vérifier le type réel d'un objet à l'exécution. On peut utiliser les transtypages, ou les opérateurs is et as.

```
using System;

namespace SyntaxeCSharp
{
```

```
class A
{}
class B : A
{}
class C : B
{}

class Programme
{
    static void Main()
    {
        // type dynamique = type de l'objet désigné par la référence
        A a=null;
        B b = null;
        C c = null;

        a = new A();  // OK type dynamique de a = A
        a = new B();  // OK type dynamique de a = B
        a = new C();  // OK type dynamique de a = C
        b = new C();  // OK type dynamique de b = C

        /*****************************************/
        // b=a;      // erreur de compilation (utiliser cast ou is/as)
        /*****************************************/

        /************************************************/
        // opérateur is : vérification dynamique de type
        /************************************************/

        a = new C();
        if (a is B)
        {
                 /* (a is B) booléen qui indique si le type
                 dynamique de a est B ou d'une sous-classe de B */

                 // ici (a is B)==true puisque a est de type dynamique C
        }
        if (a is C) { // (a is C) == true
        }

        a= null;
        if (a is C){ // (a is null)==false
        }

        /************************************************/
        // operateur as : conversion dynamique de type
        /************************************************/

        a = new B();

        b = a as B;   /// equivaut à if(a is B) { b = (B)a;} else {b = null;}

        if(b!=null){
                 // a désignait un objet compatible avec B, b est sa référence
        }

        c = a as C; // (a as C)==null puisque a n'est pas de type dynamique C
        if(c==null){
            Console.WriteLine("a n'est pas de type C");
           // l'objet désigné par a n'est pas de type compatible avec C
        }


       /************************************************/
        // cast et exception
        // les cast impossibles déclenchent des exceptions
```

```
            /***********************************************/

            a = new B();

            try
            {
                b = (B)a;   // cast OK car le type dynamique de a est B
                c = (C)a;   // ce cast déclenche une exception interceptée ci-dessous
            }
            catch(Exception e)
            {
                Console.WriteLine(e.ToString());
            }
            }
        }
    }
```

9.3. Accès aux membres non virtuels hérités

Après dérivation, un objet peut disposer de plusieurs méthodes non virtuelles de même nom. Dans l'exemple ci-dessous, un objet de la classe C dispose de 3 méthodes f(). Par défaut, une référence de type A ne permet d'atteindre que la méthode de la classe A (puisque f n'est pas virtuelle). Inversement, avec une référence de type C, on appelle la méthode de la classe C. Pour pouvoir utiliser les autres méthodes héritées, il faut transtyper.

```
using System;
namespace SyntaxeCSharp
{
    class A
    {
        public void f() { Console.WriteLine("A.f()"); }
    }
    class B : A
    {
        new public void f() { Console.WriteLine("B.f()"); }
    }
    class C : B
    {
        new public void f() { Console.WriteLine("C.f()"); }
    }

    class Programme
    {
        static void Main()
        {
            C c = new C();

            c.f();

            B b = c;
            b.f();

            ((B)c).f(); // appelle la méthode de B en raison du cast

            ((A)c).f(); // appelle la méthode de A en raison du cast
        }
    }
}
```

```
Sortie Console
C.f()
B.f()
B.f()
A.f()
Appuyez sur une touche pour
continuer...
```

9.4. Désactiver le polymorphisme dans une hiérarchie de classes

En C++, dès lors qu'une méthode est virtuelle, toutes les surdéfinitions de cette méthode dans les sous-classes disposent nécessairement du mécanisme de recherche dynamique à l'exécution. Ce mécanisme ne peut pas être contraint ou désactivé. En C#, on peut contrôler les méthodes pour lesquelles le mécanisme de recherche

dynamique à l'exécution s'applique.

Dans l'exemple ci-dessous, la méthode B.g() n'est plus virtuelle. Le mot clé new indique que le mécanisme ne doit pas s'appliquer pour les appels de cette méthode sur un objet B. Alors que la méthode f() est toujours virtuelle. On peut même regrouper les mots clé new et virtual. Voir le comportement de l'appel de la méthode f() selon qu'il s'applique à un objet B, C ou D et selon que la référence est de type A, B ou C.

```
using System;
namespace SyntaxeCSharp
{
    class A
    {
        virtual public void f() { Console.WriteLine("A.f()"); }
        virtual public void g() { Console.WriteLine("A.g()"); }
    }
    class B : A
    {
        override public void f() { Console.WriteLine("B.f()"); }
        new public void g() { Console.WriteLine("B.g()"); }
    }
    class C : B
    {
        new virtual public void f() { Console.WriteLine("C.f()"); }
        new virtual public void g() { Console.WriteLine("C.g()"); }
    }
    class D : C
    {
        override public void f() { Console.WriteLine("D.f()"); }
        override public void g() { Console.WriteLine("D.g()"); }
    }
    class Programme
    {
        static void Main()
        {
            A a1 = new B();
            A a2 = new C();
            A a3 = new D();
            B b1 = new C();
            B b2 = new D();
            C c1 = new D();
            a1.f();
            a2.f();
            a3.f();
            b1.f();
            b2.f();
            c1.f();
            Console.WriteLine("-------");
            a1.g();
            a2.g();
            a3.g();
            b1.g();
            b2.g();
            c1.g();
        }
    }
}
```

```
Sortie Console
B.f()
B.f()
B.f()
B.f()
B.f()
D.f()
-------
A.g()
A.g()
A.g()
B.g()
B.g()
D.g()
Appuyez sur une
touche pour
continuer...
```

10. Délégation (delegate) et Evénements (event)

Cet aspect de .NET (que l'on retrouve donc aussi en VB.NET) apparaît comme une nouveauté par rapport à Java. Le framework fournit un mécanisme pour pouvoir déléguer des appels de fonctions à un objet. Ce mécanisme présente des similitudes avec la notion de pointeur de fonction en C/C++.

10.1. Délégation

Le mot-clé **delegate** permet de créer des classes particulières appelées *classes de délégation*. Dans l'exemple ci-dessous, la classe de délégation est **Deleg**. Une instance de cette classe est appelée *objet délégué* ou *délégué*.

Une classe de délégation dérive implicitement de la classe **System.MulticastDelegate**. Un objet délégué doit assurer des appels à des méthodes qu'on lui a indiquées.

Lors de la définition de la classe de délégation, on mentionne également la *signature* des méthodes prises en charge par la délégation. Une instance de **Deleg** permettra d'appeler des méthodes acceptant un paramètre double et retournant un int.

Ci-dessous, **Round** désigne une instance de la classe **Deleg**. On peut ensuite utiliser l'objet **Round** comme s'il s'agissait d'une fonction acceptant un paramètre double et retournant un entier. A l'instanciation, on précise quelle méthode utiliser pour la délégation. Ensuite, on peut changer dynamiquement la méthode utilisée par l'objet délégué. Les opérateurs -= et += agissent sur l'objet délégué pour retirer ou ajouter une méthode.

```
using System;
namespace SyntaxeCSharp
{
           class Program
            {
                delegate int Deleg(double x);

                static int Floor(double x) {  return (int)x;  }

                static int Ceil(double  x)  { return (int) Math.Ceiling((double) x);}

                static void Main(string[] args)
                {
                    int val;
                    Deleg Round=new Deleg(Floor);
                    val = Round(3.2);     // utilise Floor
                    Console.WriteLine(val);
                    Round -= new Deleg(Floor); // retire Floor
                    Round += new Deleg(Ceil);  // utilise Ceil
                    val = Round(3.2);
                    Console.WriteLine(val);
                }
           }
}
```

```
Sortie Console
3
4
Appuyez sur une touche
pour continuer...
```

Simplification d'écriture avec C# 2

Avec la deuxième version de C#, le compilateur peut inférer certains types ou signatures. La définirion de la classe **Deleg** est identique. Seule la création des objets délégués devient un peu plus simple.

```
using System;
namespace SyntaxeCSharp
{
    class Program
    {
          delegate int Deleg(double x);
          static int Floor(double x) { return (int)x; }
          static int Ceil(double x) { return (int)Math.Ceiling((double)x); }

          static void Main(string[] args)
          {
               int val;
               Deleg Round = Floor;
               val = Round(3.2);     // utilise Floor
```

```
            Console.WriteLine(val);
            Round -= Floor;       // retire Floor
            Round += Ceil;        // utilise Ceil
            val = Round(3.2);
            Console.WriteLine(val);
        }
    }
}
```

10.2. Plusieurs appels pour un délégué

Un objet délégué peut prendre en charge l'appel d'une méthode mais aussi d'une liste de méthodes. Dans l'exemple ci-dessous, une instance de la classe **`CallBack`** permet d'appeler plusieurs fonctions.

```
using System;
namespace SyntaxeCSharp
{
    class Program
    {
        delegate void CallBack();

        public static void CB1()
        {
            Console.WriteLine("CB1()");
        }

        public static void CB2()
        {
            Console.WriteLine("CB2()");
        }
        static void Main(string[] args)
        {
            CallBack fonctionCB = CB1;

            Console.WriteLine("----------------------");
            fonctionCB(); // appel de CB1()

            Console.WriteLine("----------------------");

            fonctionCB += CB2;  // ajout de CB2()
            fonctionCB += CB1;  // ajout de CB1()

            fonctionCB(); // appel de CB1()->CB2()->CB1()
            Console.WriteLine("----------------------");
            fonctionCB -= CB2; // retrait de CB2()

            fonctionCB(); // appel de CB1()->CB1()
            Console.WriteLine("----------------------");
        }
    }
}
```

10.3. Appels d'une méthode d'une classe par un délégué

Les exemples précédents de délégation utilisaient des méthodes statiques. Un délégué peut aussi invoquer une méthode d'une classe mais il faut alors préciser sur quel objet elle devra être appelée. Ci-dessous, l'objet fonctionCB peut déclencher l'appel de a.CB2().

```
using System;
namespace SyntaxeCSharp
{
    class A
    {
        private int _val;
```

```
        public A(int v) { _val = v; }
        public void CB2()
        {
            Console.WriteLine("A.CB2() avec A._val = {0}",_val);
        }
    }
    class Program
    {
        delegate void CallBack();

        public static void CB1()
        {
            Console.WriteLine("CB1()");
        }

        static void Main(string[] args)
        {
            A a = new A(13);    // instance de A

            CallBack fonctionCB = a.CB2;
            Console.WriteLine("----------------------");
            fonctionCB();   // a.CB2()
            Console.WriteLine("----------------------");
            fonctionCB += CB1;

            fonctionCB();       // a.CB2()->CB1()
            Console.WriteLine("----------------------");
        }
    }
}
```

```
Sortie Console
----------------------
A.CB2() avec A._val = 13
----------------------
A.CB2() avec A._val = 13
CB1()
----------------------
Appuyez sur une touche
pour continuer...
```

10.4. Evénéments

On peut voir les événements comme une forme particulière de délégation. La similitude repose sur le fait qu'un événement peut déclencher l'appel d'une ou plusieurs méthodes (comme pour une délégation). En revanche, selon que l'on est dans la classe propriétaire de l'événement ou non, on n'a pas les même droits d'utilisation. Seule la classe propriétaire de l'événement peut déclencher des appels. Par contre, n'importe quel utilisateur peut abonner/désabonner une méthode à l'événement, c'est-à-dire ajouter ou retirer une méthode de la liste des métodes invoquées par l'événement.

Cet aspect du langage permet de simplifier la mise en oeuvre du design pattern Observer. En C#, le sujet est une classe contenant un événement. Les observateurs sont toutes les classes abonnant une de leurs méthodes à l'événement du sujet.

Dans l'exemple ci-dessous, la classe **Emetteur** a un événement appelé **EmetteurEvent**. Cet événement est déclenchable par tout objet de la classe **Emetteur**. Lorsque l'événement est déclenché, les abonnés à cet événement sont notifiés.
A noter, un événement est toujours déclaré à partir d'un type de délégation. Le type de délégation décrit la signature des méthodes pouvant s'abonner à l'événement.

L'événement ci-dessous ne déclenche que des méthodes sans paramètre ni retour.

```
class Emetteur
{
        public delegate void CallBack();  // délégation

        public event CallBack EmetteurEvent; // événement

        public void DeclencheEvenement()
         {
```

```
            // déclenche un événement = appel d'une ou plusieurs méthodes abonnées
            EmetteurEvent();
        }
}
class Program
{
        public static void CB1(){
            Console.WriteLine("Call Back 1");
        }
        public static void CB2(){
            Console.WriteLine("Call Back 2");
        }
        static void Main(string[] args)
        {
            Emetteur e = new Emetteur();

            e.EmetteurEvent += CB1;  // abonne CB1 à l'événement e.EmetteurEvent
            e.EmetteurEvent += CB2; // abonne CB2 à l'événement e.EmetteurEvent

            e.DeclencheEvenement();
            // e.EmetteurEvent(); pas possible hors classe Emetteur
     }
}
```

```
Sortie Console
Call Back 1
Call Back 2
Appuyez sur une touche
pour continuer...
```

10.5. Exploitation standard des événements dans .NET

L'utilisation des événements, notamment dans les classes System.Windows.Forms, suit toujours un peu la méme architecture. Les événements sont abondamment utilisés dans les application WinForms, puisque tous les événements du système (souris, clavier, …) sont vus comme des événements dans les classes. Un gestionnaire d'événement consiste donc à abonner une méthode de notre cru à un événement donné.

Dans ce schéma là, les événements ont généralement une signature (défini par une délégation) assez standard. Les événements appellent des méthodes ayant le plus souvent deux paramètres. Le premier est une référence à l'objet qui déclecnhe l'événement (le sujet ci-après). Le second est une référence à un objet censé transporter des données associées à l'événement. Pour un événement click souris, les données transmises sont les coordonnées du click. Le second paramètre est une référence de type System.EventArgs (ou dérivé).

Si l'on reprend ce schéma, pour une application console où l'on souhaite que le sujet puisse notifier un observateur, on obtient l'application ci-après.

La classe *Sujet* est propriétaire de l'événement.
La classe *Donnees* décrit les données transmises par l'événement (dérive de *System.EventArgs*).
System.EventHandler est une délégation standard fournie par .NET qui décrit la signature d'une méthode ne retournant rien, ayant un paramètre object et un second paramètre *System.EventArgs*.

```
using System;
namespace SyntaxeCSharp
{
    class Donnees : EventArgs
    {
        private string _message;
        public Donnees(string msg){_message=msg;}
        public string Msg(){ return _message;   }
    }

   class Sujet
   {
        public event EventHandler EvtSup20;
        private int _var;
        public Sujet(int v){ _var = v; }
        public int Var  // propriété
        {
```

```
            get
            {
                return _var;
            }
            set
            {
                _var = value;
                if (_var > 20)
                {
                 EvtSup20(this, new Donnees("Message attaché transmis par sujet"));
                }
            }
        }
    }

    class Program
    {
        public static void CB(object o, EventArgs args)
          {
              Console.WriteLine("Fonction Call Back ");
              Sujet s= o as Sujet;
              if (s != null)
              {
                  Console.WriteLine("L'état de l'annonceur = {0}",s.Var);
                  Donnees d = args as Donnees;
                  if(d !=null)
                  {
                       Console.WriteLine("Message du sujet = {0}",d.Msg());
                  }
              }
          }

        public static void Main()
         {
               Sujet leSujet = new Sujet(12);
               leSujet.EvtSup20 += CB;
               leSujet.Var = 17;  //pas d'événement
               leSujet.Var = 24;  //événement -> les abonnés sont informés -> CB appelé
        }
    }
}
```

```
Sortie Console
Fonction Call Back
L'état de l'annonceur = 24
Message du sujet = Message attaché
transmis par sujet
Appuyez sur une touche pour continuer...
```

11. Les classes conteneurs

Le framework .NET fournit des classes pour stocker des données. Nous décrivons sommairement quelques classes d'usage fréquent.

11.1. Classe string

11.1.1. Les membres de la classe string

Le type string du C# est un alias du type **`System.String`** de .NET qui permet de gérer des chaînes de caractère. Il s'agit d'un type référence mais dont l'usage ressemble à celui des types valeurs. Pour le type `string`, l'affectation fait en réalité une copie des valeurs (et non une copie de références). L'autre caractéristique est que les objets de cette classe sont *immutables*. Celà signifie que les objets gardent la même valeur du début à la fin de leur vie. Toutes les opérations visant à changer la valeur de l'objet retourneront en réalité un nouvel objet.

Quelques membres de la classe **`System.String`**

Constructeurs

string(Char[] tabC) Initialise une nouvelle instance de la classe String à la valeur indiquée par un tableau de caractères Unicode.
string(Char, Int32 repete) Initialise une nouvelle instance de la classe String à la valeur indiquée par un caractère Unicode et répété un certain nombre de fois.

string(Char[], Int32 index, Int32 count) Initialise une nouvelle instance de la classe String à la valeur indiquée par un tableau de caractères Unicode, un point de départ pour le caractère dans ce tableau et une longueur.

```
class Program
    {
        static void Main(string[] args)
        {
            string s1 = "abc";
            string s2 = new string('*', 7);
            string s3 = new string(new char[] { 'e', 'f', 'g' });

            Console.WriteLine(s1);        // abc
            Console.WriteLine(s2);        // *******
            Console.WriteLine(s3);        // efg
        }
    }
```

Champs publics

Nom	Description
Empty	**Représente la chaîne vide. Ce champ est en lecture seule.**

Propriétés publiques

Nom	Description
indexeur **[]**	Obtient le caractère à une position spécifiée dans cette instance.
Length	Obtient le nombre de caractères dans cette instance.

Méthodes publiques

public bool **EndsWith**(string value) rend vrai si la chaîne se termine par value
public bool **StartsWith**(string value) rend vrai si la chaîne commence par value

public virtual bool **Equals**(object obj) rend vrai si la chaînes est égale à obj – équivalent chaîne==obj

public int **IndexOf**(string value, int startIndex) rend la première position dans la chaîne de la chaîne value - la recherche commence à partir du caractère n° startIndex

public int **IndexOf**(char value, int startIndex) idem mais pour le caractère value

public string **Insert**(int startIndex, string value) insère la chaîne value dans chaîne en position startIndex

public string **Remove**(int startIndex,int count) supprime count caractères à partir de startIndex

public static string **Join(**string separator,string[] value) méthode de classe - rend une chaîne de caractères, résultat de la concaténation des valeurs du tableau value avec le séparateur separator

public int **LastIndexOf**(string value, int startIndex, int count) idem indexOf mais rend la dernière position au lieu de la première

```
public int LastIndexOf(char value, int startIndex,int count)

public string Replace(char oldChar, char newChar) rend une chaîne copie de la chaîne
                    courante où le caractère oldChar a été remplacé par le caractère
                    newChar

public string[] Split(char[] separator) la chaîne est vue comme une suite de champs
                    séparés par les caractères présents dans le tableau separator. Le
                    résultat est le tableau de ces champs

public string Substring(int startIndex, int length) sous-chaîne de la chaîne courante
                    commençant à la position startIndex et ayant length caractères

public string ToLower() rend la chaîne courante en minuscules

public string ToUpper() rend la chaîne courante en majuscules

public string Trim(char[] jeu ) supprime toutes les occurences d'un jeu de caractères
```

```
static void Main(string[] args)
{
            string s1 = "chaîne de caractères";

            Console.WriteLine(s1.Contains("car"));  // True
            Console.WriteLine(s1.StartsWith("ch")); // True
            Console.WriteLine(s1.IndexOf('c'));     // 0
            Console.WriteLine(s1.LastIndexOf('c')); // 14

            Console.WriteLine(s1.Substring(4, 7));  // ne de c


}
```

11.1.2. Le formattage de types numériques

La représentation des valeurs numériques par des chaînes de caractères est décrite dans l'interface **IFormattable**, c'est à dire une classe disposant d'une méthode `ToString()` avec deux paramètres

```
          interface IFormattable

          {          string ToString(string format,IFormatProvider fprovider)

          }
```

format : chaîne fournissant des instructions de formattage constituée d'une lettre avec un digit optionnel

fprovider : objet permettant de réaliser les instructions de formattage selon une culture donnée

Le programme ci-dessous affiche un prix avec au plus deux chiffres après la virgule. Dans la culture en-GB (english Great Britain) , le prix est affiché en livres. Le prix est affiché en dollars dans la culture en-US (english US).

```
£3.50
$3.50
Appuyez sur une touche pour continuer...
```

```
          static void Main(string[] args)
          {
            double prix = 3.5;
            CultureInfo culture = CultureInfo.GetCultureInfo("en-GB");
            Console.WriteLine(prix.ToString("C2", culture));
            culture = CultureInfo.GetCultureInfo("en-US");
            Console.WriteLine(prix.ToString("C2", culture));
          }
```

Chaîne de formattage des types numériques (Lettre + digit)

G ou g = général F = fixed point N = fixed point avec séparateur de groupe

E = notation exp P = pourcentage X ou x =hexadécimal

```
class Program
    {
        static void Main(string[] args)
        {
           double val1 = 1.23456;
            double val2 = 12340;
            double val3 = 0.0001234;
            double val4 = 1203.01234;
            double val5 = 0.1234;

            CultureInfo fr = CultureInfo.GetCultureInfo("fr-FR");

            Console.WriteLine("----------------");
            // G ou g : Général
            Console.WriteLine("G");
            Console.WriteLine(val1.ToString("G", fr));
            Console.WriteLine(val2.ToString("G", fr));
            Console.WriteLine(val3.ToString("G", fr));
            Console.WriteLine("----------------");
            // "G3" = limité à 3 digits au total (passe en notation
            // exp au besoin)
            Console.WriteLine(val1.ToString("G3", fr));
            Console.WriteLine(val2.ToString("G3", fr));

            Console.WriteLine("----------------");
            // F : Fixed point
            // F2 arrondit à 2 décimales
            Console.WriteLine("F");
            Console.WriteLine(val1.ToString("F2", fr));
            Console.WriteLine(val3.ToString("F2", fr));

            Console.WriteLine("----------------");
            // N : Fixed point + séparateur de groupe
            // N2 limite à deux décimales
            Console.WriteLine("N");
            Console.WriteLine(val1.ToString("N2", fr));
            Console.WriteLine(val2.ToString("N2", fr));
            Console.WriteLine(val4.ToString("N2", fr));

            Console.WriteLine("----------------");
            Console.WriteLine("E");
            // E : notation exp (6 digits par défaut après virgule)
            // E4 : 4 digits après la virgule
            Console.WriteLine(val4.ToString("E", fr));
            Console.WriteLine(val4.ToString("E4", fr));

            Console.WriteLine("----------------");
            // P : pourcentage
            Console.WriteLine("P");
            Console.WriteLine(val5.ToString("P", fr));
            Console.WriteLine(val5.ToString("P1", fr));

            Console.WriteLine("----------------");
            // X ou x :hexadécimal
            Console.WriteLine("X");
            Console.WriteLine(30.ToString("X"));

        }
    }
```

```
----------------
G
1,23456
12340
0,0001234
----------------
1,23
1,23E+04
----------------
F
1,23
0,00
----------------
N
1,23
12 340,00
1 203,01
----------------
E
1,203012E+003
1,2030E+003
----------------
P
12,34 %
12,3 %
----------------
X
1E
Appuyez sur une
touche pour
continuer...
```

11.2. Les tableaux

En C#, les tableaux sont des objets de *type référence*, compatibles avec le type de base **`Array`**. Le type **`Array`** est une classe abstraite que seul le compilateur peut dériver et implémenter. Pour construire des objets tableaux, on utilise les constructions décrites ci-dessous.

Il faut retenir que les tableaux générés ainsi ne sont pas redimensionnables. Dès lors qu'il est nécessaire de changer de dimension, les méthodes utilisées re-créent de nouveaux tableaux.

11.2.1. Tableau unidimensionnel

Déclaration des tableaux (à une dimension):

```
int[] tab;           // référence à un tableau (ausun objet n'est créé)
```

Instanciation : car un tableau est aussi vu comme un objet implémentant **`System.Array`**

```
int[] t1,t2;         // déclaration de références
int[] t3={2,6};      // déclaration + instanciation + initialisation
t1=new int[3];       // instanciation
t2=new int[]{1,2,3}; // instanciation + initialisation
```

Les tableaux sont compatibles avec le type **`System.Array`** (qui est classe de base)

```
int[] tab=null;
tab = new int[3];
Console.WriteLine(tab.GetType());
System.Array t=tab;  // t est une référence à tab
t.SetValue(3, 0);    // modifie de fait tab[0]
```

11.2.2. Parcours d'un tableau

`Length` = propriété fournissant le nombre d'éléments d'un tableau (toutes dimensions confondues)

```
int[] tab = new int[3];
for (int i = 0; i < tab.Length; i++)
{
          tab[i]=i+1;
          Console.WriteLine(tab[i]);
}
```

`foreach` : autre façon de parcourir (accès en lecture seule)

```
int[] tab = new int[3];
foreach (int val in tab)
{
Console.WriteLine(val);
}
```

11.2.3. Tableau à plusieurs dimensions

```
int[, ,] cube = new int[3, 4, 3];         // tableau à 3 dimensions
cube[0,0,0] = 1;                          // écriture dans ce tableau
```

La propriété **cube.Length** fournit alors le nombre d'éléments = 36

```
int[, ,] cube = new int[3, 4, 3];
cube[0,0,0] = 1;

Console.WriteLine(cube.Length);           // 36
Console.WriteLine(cube.GetLength(0));     // taille dans la première dimension
Console.WriteLine(cube.Rank);             // nombre de dimensions = 3
```

11.2.4. Tableaux déchiquetés (jagged arrays)

```
int[][] monTab;
monTab = new int[4][];          // monTab est un tableau
                                // pouvant stocker 4 tableaux d'int
monTab[0] = new int[4];
monTab[1] = new int[2];
monTab[2] = new int[8];
monTab[3] = new int[3];
Console.WriteLine(monTab[0][0]);
```

11.2.5. Tableaux hétérogènes

Puisque tout peut être vu comme étant de type object :

```
object[] tab = new object[3];
tab[0] = 12;
tab[1] = "toto";
tab[2] = 12.4F;
```

11.2.6. Copie de tableaux

Première version, utilise la méthode **`Array.CopyTo()`**

```
int[] t1 = {1,2,3};
int[] t2 = new int[t1.Length];

t1.CopyTo(t2,0);
```

Seconde version, utilise la méthode **`Clone()`** qui pour les objets clonables (implémentant `ICloneable`), renvoit un objet copie

```
int[] t1 = {1,2,3};
int[] t2;

t2= (int[]) t1.Clone();  // t1.Clone() est de type object, d'où cast
```

11.2.7. Membres de la classe Array

`Array` est une classe abstraite implémentant certaines interfaces

```
namespace System
{
        public abstract class Array : ICloneable, IList,
                ICollection,IEnumerable
        { }
}
```

Propriétés

`public int` **`Length`** `{get;}` nombre total d'éléments du tableau, quel que soit son nombre de dimensions

`public long` **`LongLength`** `{get;}` idem mais sur 64 bits

`public int` **`Rank`** `{get;}` nombre total de dimensions du tableau

Méthodes

`public static int` **`BinarySearch<T>`**`(T[] tableau,T value)` rend la position de value dans un **tableau trié** unidimensionnel

`public static int` **`BinarySearch<T>`**`(T[] tableau,int index, int length, T value)` idem mais cherche dans tableau trié à partir de la position index et sur length éléments

public static void **Clear**(Array tableau, int index, int length) met les length éléments de tableau commençant au n° index à 0 si numériques, false si booléens, null si références

public static void **Copy**(Array source, Array destination, int length) copie length éléments de source dans destination

public int **GetLength**(int i) nombre d'éléments de la dimension n° i du tableau

public int **GetLowerBound**(int i) indice du 1er élément de la dimension n° i

public int **GetUpperBound**(int i) indice du dernier élément de la dimension n° i

public static int **IndexOf<T>**(T[] tableau, T valeur) rend la position de valeur dans tableau ou -1 si valeur n'est pas trouvée.

public static void **Resize<T>**(ref T[] tableau, int n) redimensionne tableau à n éléments. Les éléments déjà présents sont conservés.

public static void **Sort<T>**(T[] tableau, IComparer<T> comparateur) trie tableau selon un ordre défini par comparateur.

11.3. Les conteneurs génériques.

Le C# permet la définition de classes paramétrées en type. Celà signifie que certains types peuvent être choisis au moment de l'instanciation. Du fait que certains types sont paramétrables, on parle de classe générique.

Sans rentrer dans les détails, on peut fournir un exemple simple de classe générique. On représente des paires de valeurs. Le type des valeurs de la paire est un paramètre de type de la classe.

```
namespace ConsoleGenerique
{
    class Paire<T>  // classe générique : T est yb type paramétrable
    {
        private T _premier;  // les deux attributs sont du même type
        private T _second;

        public Paire(T prem, T sec)   // constructeur
        {
            _premier = prem;
            _second = sec;
        }

        public override string ToString()
        {
            return "(" + _premier.ToString() + "," + _second.ToString() + ")";
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
           // paire d'entiers
             Paire<int> p1= new Paire<int>(7,2);
             Console.WriteLine(p1.ToString());

           // paire de chaînes
             Paire<string> p2 = new Paire<string>("premier", "second");
             Console.WriteLine(p2.ToString());

           // paire de paires de double (puisque Paire<double> est aussi un type)
             Paire<Paire<double>> p3;
           p3 = new Paire<Paire<double>>(new Paire<double>(1.2, 2.3),
                                         new Paire<double>(3.5, 4.8));
             Console.WriteLine(p3.ToString());

        }
```

Sortie console

```
(7,2)
(premier,second)
((1,2,2,3),(3,5,4,8))
```

```
    }
}
```

11.3.1. Quelques interfaces de conteneurs : IEnumerable, ICollection, Ilist, IDictionary

Les classes conteneurs sont toutes les classes fournies par le framework .NET qui permettent de stocker des données. Le premier type de conteneur utilisé est le tableau (type dérivé de **`Array`**). D'autres classes génériques permettent de stocker des données. Les conteneurs se distinguent par le type de service qu'ils rendent.

Interface `IEnumerable<T>`

Le comportement minimum d'une structure de donnée est d'implémenter **`IEnumerable`**. C'est à dire que la structure de donnée peut fournir un énumérateur permettant de parcourir les données.

Dès lors qu'un type implémente **`IEnumerable`**, on peut utiliser l'instruction **`foreach`** pour parcourir ses éléments.

```
    public interface IEnumerable
    {
          // Retourne: Objet System.Collections.IEnumerator pouvant
          // être utilisé pour itérer au sein de la collection.
        IEnumerator GetEnumerator();
    }
```

```
   public interface IEnumerator
    {
        object Current { get; }  // fournit élément courant.

        // Avance l'énumérateur à l'élément suivant de la collection.
        // Retourne: true si l'énumérateur a pu avancer
        bool MoveNext();

     //  remet l'énumérateur à sa position initiale (avant premier élément)
        void Reset();
    }
```

`Interface ICollection<T>`

Plus riche que l'interface `IEnumerable`, l'interface **`ICollection`** est décrite ci-dessous

```
public namespace System.Collections.Generic

{
    interface ICollection<T> : IEnumerable<T>, IEnumerable
    {

        int Count { get; } //  Nombre d'éléments

        bool IsReadOnly { get; }          //     Est en lecture seule

        void Add(T item);      //  Ajoute un élément item

        void Clear(); //     Supprime tous les éléments du

        bool Contains(T item);    //    Est-ce que item est présent

        void CopyTo(T[] array, int arrayIndex); // Copie dans un tableau
        bool Remove(T item);  // supprime l'élément item
    }
```

Interface IList<T>

Plus riche que l'interface `ICollection`, l'interface **IList<T>** permet d'atteindre des éléments par un index (comme un tableau). La classe générique **List<T>** implémente cette interface.

```
namespace System.Collections.Generic
{
    public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
    {
        T this[int index] { get; set; }  // indexeur

        int IndexOf(T item); // donne l'index de item ou -1 sinon

        void Insert(int index, T item); // insertion de item à l'index

        void RemoveAt(int index); // supprime l'élment à index
    }
}
```

Interface IDictionary<TKey,TValue>

C'est l'interface pour les structures de données (clé,valeur). On appelle parfois celà des tableaux associatifs. On peut accéder aux élements en fournissant non pas un indice, mais une clé.

```
namespace System.Collections.Generic
{

    public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,
           IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
    {
        ICollection<TKey> Keys { get; }              // fournit la collection des clés
        ICollection<TValue> Values { get; }          // les valeurs

        TValue this[TKey key] { get; set; }  // valeur de clé key en lecture/écriture


        void Add(TKey key, TValue value);  // ajoute une valeur pour la clé key

        bool ContainsKey(TKey key);  // recherche présence de la clé
        bool Remove(TKey key);       // supprime la clé
        bool TryGetValue(TKey key, out TValue value);
    }
}
```

11.3.2. List<T>

La classe générique **List<T>** permet de générer des tableaux (et non des listes chaînées) dont la taille peut être dynamiquement modifiée par l'ajout ou la suppression. Rappelons que les tableaux (`Array`) eux ne sont pas redimensionnables. Cette classe générique implémente l'interface **IList<T>**.

```
namespace System.Collections.Generic
{
       public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
                                IList, ICollection, IEnumerable
          {
           public List();  // tableau vide de capacité par défaut
              public List(IEnumerable<T> collection);  // tableau initialisé avec collection
              public List(int capacity);  // tableau vide de capacité fournie

              public int Capacity { get; set; }  // taille mémoire disponible
```

```
                                        // avant reallocation

        public int Count { get; }   // nombre d'éléments
        public T this[int index] { get; set; }  // indexeur
        public void Add(T item);

        public void AddRange(IEnumerable<T> collection); // ajout en fin

        public int BinarySearch(T item); // recherche dans collection triée
        public int BinarySearch(T item, IComparer<T> comparer); // idem
        public int BinarySearch(int index, int count, T item, IComparer<T> comparer);

        public void Clear();  // supprime tous les éléments
        public bool Contains(T item); // teste la présence de item

        public void CopyTo(T[] array);  // copie vers array
        public void CopyTo(T[] array, int arrayIndex); // à partir de arrayIndex
        public void CopyTo(int index, T[] array, int arrayIndex, int count);

        public bool Exists(Predicate<T> match); // teste le prédicat sur la collection

        public T Find(Predicate<T> match); // premiere occurrence qui vérifie match
         public List<T> FindAll(Predicate<T> match); // éléments qui vérifient match
        public int FindIndex(Predicate<T> match); // index du premier qui matche
        public int FindIndex(int startIndex, Predicate<T> match);
        public int FindIndex(int startIndex, int count, Predicate<T> match);
        public T FindLast(Predicate<T> match);
        public int FindLastIndex(Predicate<T> match);
        public int FindLastIndex(int startIndex, Predicate<T> match);
        public int FindLastIndex(int startIndex, int count, Predicate<T> match);

        public void ForEach(Action<T> action); // execute Action<T> sur tous

        public List<T>.Enumerator GetEnumerator(); // obtient un Enumerator
        public List<T> GetRange(int index, int count); // retourne une partie

        public int IndexOf(T item);  // indice de item ou -1
        public int IndexOf(T item, int index);  // indice de item à partir de index
        public int IndexOf(T item, int index, int count);

        public void Insert(int index, T item);  // insère item à l'indice index

        public void InsertRange(int index, IEnumerable<T> collection);

        public int LastIndexOf(T item);
        public int LastIndexOf(T item, int index);
        public int LastIndexOf(T item, int index, int count);

        public bool Remove(T item);
        public int RemoveAll(Predicate<T> match);
        public void RemoveAt(int index);
        public void RemoveRange(int index, int count);

        public void Reverse();  // inverse l'ordre des éléments
        public void Reverse(int index, int count); // idem dans l'intervalle spécifié

        public void Sort();  // trie
        public void Sort(IComparer<T> comparer); // trie selon comparateur fourni
        public void Sort(int index, int count, IComparer<T> comparer);

        public T[] ToArray();  // fournit les éléments dans un tableau
    }
```

11.3.3. LinkedList<T>

Les objets de type **LinkedList<T>** sont des listes doublement chaînées d'éléments de type T.

Cette classe n'implémente pas `IList<T>` puisque l'on ne peut pas accéder efficacement directement à un élément.

Les éléments sont stockés dans des noeuds dont la structure est décrite ci-dessous.

```
namespace System.Collections.Generic
{
    public sealed class LinkedListNode<T>
    {
        public LinkedListNode(T value);
        public LinkedList<T> List { get; }
        public LinkedListNode<T> Next { get; }
        public LinkedListNode<T> Previous { get; }
        public T Value { get; set; }
    }
}
```

Les méthodes de la classe `LinkedList<T>` sont décrites ci-dessous

```
namespace System.Collections.Generic
{

    public class LinkedList<T> : ICollection<T>, IEnumerable<T>,
                   ICollection, IEnumerable, ISerializable,
                   IDeserializationCallback
    {
        public LinkedList(); // crée Liste vide
        public LinkedList(IEnumerable<T> collection);
        protected LinkedList(SerializationInfo info, StreamingContext context);

        public int Count { get; }  // nombre d'éléments
        public LinkedListNode<T> First { get; } // premier et dernier noeud
        public LinkedListNode<T> Last { get; }

        public void AddAfter(LinkedListNode<T> node,
                                    LinkedListNode<T> newNode);
        public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);
        public void AddBefore(LinkedListNode<T> node,
                                    LinkedListNode<T> newNode);
        public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);
        public void AddFirst(LinkedListNode<T> node);
        public LinkedListNode<T> AddFirst(T value);
        public void AddLast(LinkedListNode<T> node);
        public LinkedListNode<T> AddLast(T value);

        public void Clear(); // supprime tous les noeuds
        public bool Contains(T value); // indique présence de value

        public void CopyTo(T[] array, int index);
        public LinkedListNode<T> Find(T value);
        public LinkedListNode<T> FindLast(T value);
        public LinkedList<T>.Enumerator GetEnumerator();

        public void Remove(LinkedListNode<T> node);
        public bool Remove(T value);
        public void RemoveFirst();  // supprime le premier noeud
        public void RemoveLast();     // supprime le dernier
    }
}
```

11.3.4. Classe Dictinary<TKey,TValue> (tableau associatif (clé,valeur))

Classe générique de tableaux associatifs implémentant l'interface `IDictionary<key,value>`.

```
namespace System.Collections.Generic
{

    public class Dictionary<TKey, TValue> : IDictionary<TKey, TValue>,
          ICollection<KeyValuePair<TKey, TValue>>,
          IEnumerable<KeyValuePair<TKey, TValue>>,
          IDictionary, ICollection, IEnumerable, Iserializable,
          IDeserializationCallback
    {
        public Dictionary();
        public Dictionary(IDictionary<TKey, TValue> dictionary);

        public int Count { get; }
        public Dictionary<TKey, TValue>.KeyCollection Keys { get; }
        public Dictionary<TKey, TValue>.ValueCollection Values { get; }

        public TValue this[TKey key] { get; set; }
        public void Add(TKey key, TValue value);
        public void Clear();
        public bool ContainsKey(TKey key);
        public Dictionary<TKey, TValue>.Enumerator GetEnumerator();
        public bool Remove(TKey key);
        public bool TryGetValue(TKey key, out TValue value);

          }
}
```

11.4. Table des matières